11-22-04

IFW

PTO/SB/21 (09-04)

TRANSMITTAL FORM

(to be used for all correspondence after initial filing)

Total Number of Pages in This Submission	9

Application Number	10/655,692
Filing Date	September 5, 2003
First Named Inventor	Hashimoto, Akiyoshi
Art Unit	2185
Examiner Name	Unassigned
Attorney Docket Number	16869S-093900US

ENCLOSURES (Check all that apply)

- ☒ Fee Transmittal Form
  - ☐ Fee Attached
- ☐ Amendment/Reply
  - ☐ After Final
  - ☐ Affidavits/declaration(s)
- ☐ Extension of Time Request
- ☐ Express Abandonment Request
- ☐ Information Disclosure Statement
- ☐ Certified Copy of Priority Document(s)
- ☐ Reply to Missing Parts/ Incomplete Application
  - ☐ Reply to Missing Parts under 37 CFR 1.52 or 1.53
- ☐ Drawing(s)
- ☐ Licensing-related Papers
- ☒ Petition to Make Special
- ☐ Petition to Convert to a Provisional Application
- ☐ Power of Attorney, Revocation Change of Correspondence Address
- ☐ Terminal Disclaimer
- ☐ Request for Refund
- ☐ CD, Number of CD(s) _____
  - ☐ Landscape Table on CD
- ☐ After Allowance Communication to TC
- ☐ Appeal Communication to Board of Appeals and Interferences
- ☐ Appeal Communication to TC **(Appeal Notice, Brief, Reply Brief)**
- ☐ Proprietary Information
- ☐ Status Letter
- ☒ Other Enclosure(s) (please identify below):
  - Return Postcard
  - Six (6) cited references

Remarks The Commissioner is authorized to charge any additional fees to Deposit Account 20-1430.

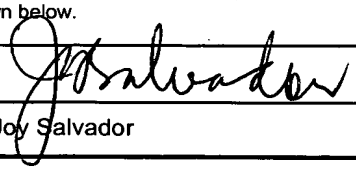
SIGNATURE OF APPLICANT, ATTORNEY, OR AGENT

Firm Name	Townsend and Townsend and Crew LLP		
Signature			
Printed name	Chun-Pok Leung		
Date	November 18, 2004	Reg. No.	41,405

CERTIFICATE OF TRANSMISSION/MAILING

Express Mail Label: **EV 530886429 US**

I hereby certify that this correspondence is being deposited with the United States Postal Service with "Express Mail Post Office to Address" service under 37 CFR 1.10 on this date **November 18, 2004** and is addressed to: Commissioner for Patents, P.O. Box 1450, Alexandria, VA 22313-1450 on the date shown below.

Signature			
Typed or printed name	Joy Salvador	Date	November 18, 2004

FEE TRANSMITTAL for FY 2005

Effective 10/01/2004. Patent fees are subject to annual revision.

☒ Applicant claims small entity status. See 37 CFR 1.27

TOTAL AMOUNT OF PAYMENT ($) **130.00**

Complete if Known	
Application Number	10/655,692
Filing Date	September 5, 2003
First Named Inventor	Hashimoto, Akiyoshi
Examiner Name	Unassigned
Art Unit	2185
Attorney Docket No.	16869S-093900US

METHOD OF PAYMENT *(check all that apply)*

☐ Check ☐ Credit Card ☐ Money Order ☐ Other ☐ None

☒ Deposit Account:

Deposit Account Number: 20-1430

Deposit Account Name: Townsend and Townsend and Crew LLP

The Director is authorized to: *(check all that apply)*

☒ Charge fee(s) indicated below ☒ Credit any overpayments

☒ Charge any additional fee(s) or any underpayment of fee(s)

☐ Charge fee(s) indicated below, **except for the filing fee** to the above-identified deposit account.

FEE CALCULATION

1. BASIC FILING FEE

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description	Fee Paid
1001	790	2001	395	Utility filing fee	
1002	350	2002	175	Design filing fee	
1003	550	2003	275	Plant filing fee	
1004	790	2004	395	Reissue filing fee	
1005	160	2005	80	Provisional filing fee	

SUBTOTAL (1) ($)0.00

2. EXTRA CLAIM FEES FOR UTILITY AND REISSUE

			Extra Claims		Fee from below		Fee Paid
Total Claims		-** =		X		=	
Independent Claims		-** =		X		=	
Multiple Dependent				X		=	

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description
1202	18	2202	9	Claims in excess of 20
1201	88	2201	44	Independent claims in excess of 3
1203	300	2203	150	Multiple dependent claim, if not paid
1204	88	2204	44	** Reissue independent claims over original patent
1205	18	2205	9	** Reissue claims in excess of 20 and over original patent

SUBTOTAL (2) ($)0.00

**or number previously paid, if greater; For Reissues, see above

FEE CALCULATION (continued)

3. ADDITIONAL FEES

Large Entity Fee Code	Large Entity Fee ($)	Small Entity Fee Code	Small Entity Fee ($)	Fee Description	Fee Paid
1051	130	2051	65	Surcharge - late filing fee or oath	
1052	50	2052	25	Surcharge - late provisional filing fee or cover sheet.	
1053	130	1053	130	Non-English specification	
1812	2,520	1812	2,520	For filing a request for *ex parte* reexamination	
1804	920*	1804	920*	Requesting publication of SIR prior to Examiner action	
1805	1,840*	1805	1,840*	Requesting publication of SIR after Examiner action	
1251	110	2251	55	Extension for reply within first month	
1252	430	2252	215	Extension for reply within second month	
1253	980	2253	490	Extension for reply within third month	
1254	1,530	2254	765	Extension for reply within fourth month	
1255	2,080	2255	1,040	Extension for reply within fifth month	
1401	340	2401	170	Notice of Appeal	
1402	340	2402	170	Filing a brief in support of an appeal	
1403	300	2403	150	Request for oral hearing	
1451	1,510	1451	1,510	Petition to institute a public use proceeding	
1452	110	2452	55	Petition to revive – unavoidable	
1453	1,330	2453	665	Petition to revive – unintentional	
1501	1,370	2501	685	Utility issue fee (or reissue)	
1502	490	2502	245	Design issue fee	
1503	660	2503	330	Plant issue fee	
1460	130	1460	130	Petitions to the Commissioner	130
1807	50	1807	50	Processing fee under 37 CFR 1.17(q)	
1806	180	1806	180	Submission of Information Disclosure Stmt	
8021	40	8021	40	Recording each patent assignment per property (times number of properties)	
1809	790	2809	395	Filing a submission after final rejection (37 CFR § 1.129(a))	
1810	790	2810	395	For each additional invention to be examined (37 CFR § 1.129(b))	
1801	790	2801	395	Request for Continued Examination (RCE)	
1802	900	1802	900	Request for expedited examination of a design application	

Other fee (specify) _____

*Reduced by Basic Filing Fee Paid **SUBTOTAL (3)** ($)130.00

SUBMITTED BY				Complete *(if applicable)*	
Name (Print/Type)	Chun-Pok Leung	*Registration No. (Attorney/Agent)*	41,405	*Telephone*	650-326-2400
Signature				*Date*	November 18, 2004

WARNING: Information on this form may become public. Credit card information should not be included on this form. Provide credit card information and authorization on PTO-2038.

60359826 v1

11/22/04

IFW

PATENT
Attorney Docket No.: 16869S-093900US
Client Ref. No.: W1145-01EF

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re application of:

AKIYOSHI HASHIMOTO

Application No.: 10/655,692

Filed: September 5, 2003

For: FILE SERVER SYSTEM

Customer No.: 20350

Examiner: Unassigned

Technology Center/Art Unit: 2185

Confirmation No.: 6495

PETITION TO MAKE SPECIAL FOR NEW APPLICATION UNDER M.P.E.P. § 708.02, VIII & 37 C.F.R. § 1.102(d)

Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

Sir:

This is a petition to make special the above-identified application under MPEP § 708.02, VIII & 37 C.F.R. § 1.102(d). The application has not received any examination by an Examiner.

(a) The Commissioner is authorized to charge the petition fee of $130 under 37 C.F.R. § 1.17(i) and any other fees associated with this paper to Deposit Account 20-1430.

11/24/2004 MBEYENE1 00000094 201430 10655692
01 FC:1460 130.00 DA

(b) All the claims are believed to be directed to a single invention. If the Office determines that all the claims presented are not obviously directed to a single invention, then Applicants will make an election without traverse as a prerequisite to the grant of special status.

(c) Pre-examination searches were made of U.S. issued patents, including a classification search and a computer database search. The searches were performed on or around September 15, 2004, and were conducted by a professional search firm, Kramer & Amado, P.C. The classification search covered Class 709 (subclasses 203, 219, and 225), Class 711 (subclasses 111 and 112), and Class 713 (subclasses 165, 200, 201, and 202) for the U.S. and foreign subclasses identified above. The computer database search was conducted on the USPTO systems EAST and WEST. The inventors further provided three references considered most closely related to the subject matter of the present application (see references #4-6 below), which were cited in the Information Disclosure Statements filed on October 9, 2003.

(d) The following references, copies of which are attached herewith, are deemed most closely related to the subject matter encompassed by the claims:

(1) U.S. Patent No. 6,122,631;

(2) U.S. Patent Publication No. 2002/0103904 A1;

(3) U.S. Patent Publication No. 2003/0023784 A1;

(4) Bakke et al., "iSCSI Naming and Discovery," available on-line at http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-name-disc-09.txt, Internet Draft of IPS Working Group, the Intenet Society (2002);

(5) Gibson et al., "File Server Scaling with Network-Attached Secure Disks," Proceedings of the 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems, Seattle, WA (1997); and

(6) VAHALIA UNIX Internals: The New Frontiers, pp. 291-313, Prentice Hall (1995).

(e) Set forth below is a detailed discussion of references which points out with particularity how the claimed subject matter is distinguishable over the references.

A. Claimed Embodiments of the Present Invention

The claimed embodiments relate to a file server that provides a plurality of networked clients with file services.

Independent claim 1 recites a file server system comprising a plurality of hard disk drives connected to a plurality of clients via a network; and a file control unit connected to the network for accepting an access request from the clients to the hard disk drives to manage the data input/output of the plurality of hard disk drives. The file control unit has configuration information with which a plurality of pieces of identification (ID) information, each identifying one of the plurality of hard disk drives, can be registered. The file control unit broadcasts a hard disk drive search message via the network. In response to the hard disk drive search message, the hard disk drive returns the ID information specifying the self hard disk drive to the file control unit. In response to the returned ID information, the file control unit establishes a setting such that the hard disk drive, which has returned the ID information, cannot communicate with devices on the network other than the file control unit.

Independent claim 11 recites a file server system comprising a plurality of switching hubs interconnected to form a network; a plurality of hard disk drives connected to clients via the network; and a file control unit. Each of the plurality of hard disk drives is connected to one of the plurality of switching hubs. The file control unit is connected to one of the plurality of switching hubs. The file control unit accepts an access request from the clients to the hard disk drives to manage a data input/output of the plurality of hard disk drives. The switching hubs perform control so that the file control unit and the plurality of clients belong to a virtual network and so that the file control unit and the plurality of hard disk drives belong to another virtual network.

One of the benefits that may be derived is securing the safety of data saved on hard disk drives when clients and hard disk drives are connected to the same LAN. By

preventing the clients or the management terminal from reading data from, or writing data to, the hard disk drive without obtaining permission from the file control unit, the system ensures data safety since data is transferred always via the file control unit. In addition, by inhibiting the clients and the management terminal from directly accessing hard disk drives in the system employing separate virtual networks, the file control unit and the hard disk drives need not have the encrypted communication function and thus the cost may be reduced.

B. Discussion of the References

None of the following references disclose a file server system in which the file control unit broadcasts a hard disk drive search message via the network; wherein, in response to the hard disk drive search message, the hard disk drive returns the ID information specifying the self hard disk drive to the file control unit; and wherein, in response to the returned ID information, the file control unit establishes a setting such that the hard disk drive, which has returned the ID information, cannot communicate with devices on the network other than the file control unit.

The references also do not teach a file server system in which the file control unit accepts an access request from the clients to the hard disk drives to manage a data input/output of the plurality of hard disk drives, wherein the switching hubs perform control so that the file control unit and the plurality of clients belong to a virtual network and so that the file control unit and the plurality of hard disk drives belong to another virtual network.

1. U.S. Patent No. 6,122,631

This reference discloses a dynamic server-managed access control for a distributed file system with a server file access token which it delivers to the distributed file system and to the client. The client uses the token in place of a standard file name. If a request to access (open) the file is received from a client with the token, the file is opened.

2. U.S. Patent Publication No. 2002/0103904 A1

This reference discloses a method and an apparatus for controlling access to files associated with a virtual server. Each virtual server is also assigned an identifier to uniquely identify that server and all files associated with that server. The server computing device retrieves the identifier assigned to the existing file. Next, the server computing device

determines whether the identifier is associated with the virtual server that generated the file access request. If the identifier is associated with the virtual server that generated the file access request, the server computing device allows access to take place.

3. U.S. Patent Publication No. 2003/0023784 A1

This reference relates to a storage system having a plurality of controllers. The assigned controller identity number is the identification number of the controller to which the disk drive unit identified at path ID and address is allocated. The processor searches the disk pool management table using the identification number in the inquiry as the search key to identify the disk drive units that can be used by the inquiring controller. See [0070] and [0084].

4. Bakke et al., "iSCSI Naming and Discovery," available on-line at http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-name-disc-09.txt, Internet Draft of IPS Working Group, the Intenet Society (2002)

This reference discloses iSCSI which is a standard that allows SCSI protocol communication to be performed on a network.

5. Gibson et al., "File Server Scaling with Network-Attached Secure Disks," Proceedings of the 1997 ACM Sigmetrics International Conference on Measurement and Modeling of Computer Systems, Seattle, WA (1997)

This reference discloses NetSCSI and NASD. This is discussed in the present specification at page 2, line 14 to page 4, line 21.

6. VAHALIA UNIX Internals: The New Frontiers, pp. 291-313, Prentice Hall (1995)

This reference relates to NFS (Network File System), which is a technology for managing data on a file basis. A computer in which files are saved is called a file server, and a computer that uses the files saved in a file server via a network is called a client. NFS is a technology that allows the user to use files saved in the file server as if they were saved in the client's disk. In practice, NFS is defined as a network communication protocol between a file server and a client.

(f) In view of this petition, the Examiner is respectfully requested to issue a first Office Action at an early date.

Respectfully submitted,

Chun-Pok Leung
Reg. No. 41,405

TOWNSEND and TOWNSEND and CREW LLP
Two Embarcadero Center, 8th Floor
San Francisco, California 94111-3834
Tel: 650-326-2400
Fax: 415-576-0300
Attachments
RL:rl
60351325 v1

W1145

IP Storage Working Group
Internet Draft
draft-ietf-ips-iscsi-19.txt
Category: standards-track

Julian Satran
Kalman Meth
IBM

Costa Sapuntzakis
Cisco Systems

Mallikarjun Chadalapaka
Hewlett-Packard Co.

Efri Zeidner
SANGate

iSCSI

IPS Working Group
Internet Draft (Informational Track)
draft-ietf-ips-iscsi-name-disc-09.txt
Expires September 2003

Mark Bakke
Cisco

Jim Hafner
John Hufferd
Kaladhar Voruganti
IBM

Marjorie Krueger
Hewlett-Packard

iSCSI Naming and Discovery

Status of this Memo

Copyright Notice

Abstract

This document provides examples of iSCSI (SCSI over TCP) name construction and discussion of discovery of iSCSI resources (targets) by iSCSI initiators. This document complements the iSCSI protocol draft. Flexibility is the key guiding principle behind this document. That is, an effort has been made to satisfy the needs of both small isolated environments, as well as large environments

requiring secure/scalable solutions.

Acknowledgements

Joe Czap (IBM), Howard Hall (Pirus), Jack Harwood (EMC), Yaron Klein (SANRAD), Larry Lamers (Adaptec), Josh Tseng (Nishan Systems) and Todd Sperry (Adaptec) have participated and made contributions during development of this document.

Table of Contents

1. iSCSI Names and Addresses

The main addressable, discoverable entity in iSCSI is an iSCSI Node. An iSCSI node can be either an initiator, a target, or both. The rules for constructing an iSCSI name are specified in [iSCSI].

This document provides examples of name construction that might be used by a naming authority.

Both targets and initiators require names for the purpose of identification, and so that iSCSI storage resources can be managed regardless of location (address). An iSCSI name is the unique identifier for an iSCSI node, and is also the SCSI device name [SAM2] of an iSCSI device. The iSCSI name is the principal object used in authentication of targets to initiators and initiators to targets. This name is also used to identify and manage iSCSI storage resources.

Furthermore, iSCSI names are associated with iSCSI nodes instead of with network adapter cards to ensure the free movement of network HBAs between hosts without loss of SCSI state information (reservations, mode page settings etc) and authorization configuration.

An iSCSI node also has one or more addresses. An iSCSI address specifies a single path to an iSCSI node and consists of the iSCSI

name, plus a transport (TCP) address which uses the following format:

<domain-name>[:<port>]

Where <domain-name> is one of:

- IPv4 address, in dotted decimal notation. Assumed if the name contains exactly four numbers, separated by dots (.), where each number is in the range 0..255.

- IPv6 address, in colon-separated hexadecimal notation, as specified in [RFC2373] and enclosed in "[" and "]" characters, as specified in [RFC2732].

- Fully Qualified Domain Name (host name). Assumed if the <domain-name> is neither an IPv4 nor an IPv6 address.

For iSCSI targets, the <port> in the address is optional; if specified, it is the TCP port on which the target is listening for connections. If the <port> is not specified, the default port 3260, assigned by IANA, will be assumed. For iSCSI initiators, the <port> is omitted.

Examples of addresses:

```
 192.0.2.2
192.0.2.23:5003
[FEDC:BA98:7654:3210:FEDC:BA98:7654:3210]
[1080:0:0:0:8:800:200C:417A]
[3ffe:2a00:100:7031::1]
[1080::8:800:200C:417A]
[1080::8:800:200C:417A]:3260
[::192.0.2.5]
mydisks.example.com
moredisks.example.com:5003
```

The concepts of names and addresses have been carefully separated in iSCSI:

- An iSCSI Name is a location-independent, permanent identifier for an iSCSI node. An iSCSI node has one iSCSI name, which stays constant for the life of the node. The terms "initiator name" and "target name" also refer to an iSCSI name.

- An iSCSI Address specifies not only the iSCSI name of an iSCSI node, but also a location of that node. The address consists of a host name or IP address, a TCP port number (for the target), and the iSCSI Name of the node. An iSCSI node can have any number of

addresses, which can change at any time, particularly if they are assigned via DHCP.

A similar analogy exists for people. A person in the USA might be:

Robert Smith
SSN+DateOfBirth: 333-44-5555 14-MAR-1960
Phone: +1 (763) 555.1212
Home Address: 555 Big Road, Minneapolis, MN 55444
Work Address: 222 Freeway Blvd, St. Paul, MN 55333

In this case, Robert's globally unique name is really his Social Security Number plus Date of Birth. His common name, "Robert Smith", is not guaranteed to be unique. Robert has three locations at which he may be reached; two Physical addresses, and a phone number.

In this example, Robert's SSN+DOB is like the iSCSI Name (date of birth is required to disambiguate SSNs that have been reused), his phone number and addresses are analogous to an iSCSI node's TCP addresses, and "Robert Smith" would be a human-friendly label for this person.

To assist in providing a more human-readable user interface for devices that contain iSCSI targets and initiators, a target or initiator may also provide an alias. This alias is a simple UTF-8 string, is not globally unique, and is never interpreted or used to identify an initiator or device within the iSCSI protocol. Its use is described further in section 2.

1.1. Constructing iSCSI names using the iqn. format

The iSCSI naming scheme was constructed to give an organizational naming authority the flexibility to further subdivide the responsibility for name creation to subordinate naming authorities. The iSCSI qualified name format is defined in [iSCSI] and contains (in order):

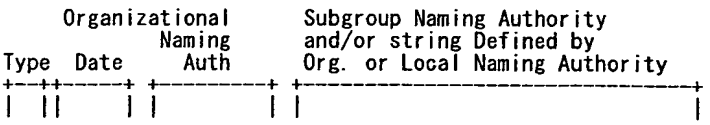
- The string "iqn."

- A date code specifying the year and month in which the organization registered the domain or sub-domain name used as the naming authority string.

- The organizational naming authority string, which consists of a valid, reversed domain or subdomain name.

- Optionally, a ':', followed by a string of the assigning organization's choosing, which must make each assigned iSCSI name unique.

The following is an example of an iSCSI qualified name from an equipment vendor:



iqn.2001-04.com.example:diskarrays-sn-a8675309

Where:

"iqn" specifies the use of the iSCSI qualified name as the authority.

"2001-04" is the year and month on which the naming authority acquired the domain name used in this iSCSI name. This is used to ensure that when domain names are sold or transferred to another organization, iSCSI names generated by these organizations will be unique.

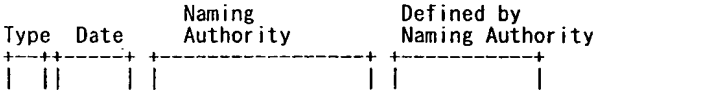
"com.example" is a reversed DNS name, and defines the organizational naming authority. The owner of the DNS name "example.com" has the sole right of use of this name as this part of an iSCSI name, as well as the responsibility to keep the remainder of the iSCSI name unique. In this case, example.com happens to manufacture disk arrays.

"diskarrays" was picked arbitrarily by example.com to identify the disk arrays they manufacture. Another product that ACME makes might use a different name, and have its own namespace independent of the disk array group. The owner of "example.com" is responsible for keeping this structure unique.

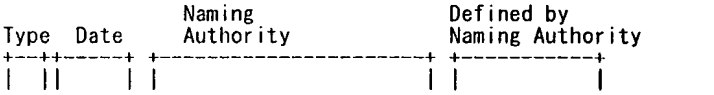
"sn" was picked by the disk array group of ACME to show that what follows is a serial number. They could have just assumed that all iSCSI Names are based on serial numbers, but they thought that perhaps later products might be better identified by something else. Adding "sn" was a future-proof measure.

"a8675309" is the serial number of the disk array, uniquely identifying it from all other arrays.

Another example shows how the ':' separator helps owners of sub-domains to keep their name spaces unique:



iqn.2001-04.com.example.storage:tape.sys1.xyz

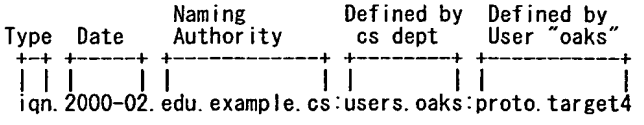


iqn.2001-04.com.example.storage.tape:sys1.xyz

Note that, except for the ':' separator, both names are identical. The first was assigned by the owner of the subdomain

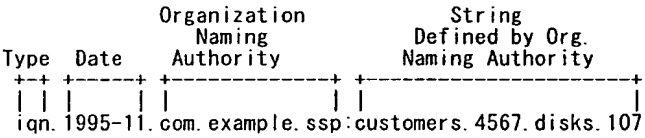
"storage.example.com"; the second was assigned by the owner of "tape.storage.example.com". These are both legal names, and are unique.

The following is an example of a name that might be constructed by an research organization:



In the above example, Professor Oaks of Example University is building research prototypes of iSCSI targets. EU's computer science department allows each user to use his or her user name as a naming authority for this type of work, by attaching "users.<username>" after the ':', and another ':', followed by a string of the user's choosing (the user is responsible for making this part unique). Professor Oaks chose to use "proto.target4" for this particular target.

The following is an example of an iSCSI name string from a storage service provider:



In this case, a storage service provider (ssp.example.com) has decided to re-name the targets from the manufacturer, to provide the flexibility to move the customer's data to a different storage subsystem should the need arise.

The SSP has configured the iSCSI Name on this particular target for one of its customers, and has determined that it made the most sense to track these targets by their Customer ID number and a disk number. This target was created for use by customer #4567, and is the 107th target configured for this customer.

Note that when reversing these domain names, the first component (after the "iqn.") will always be a top-level domain name, which includes "com", "edu", "gov", "org", "net", "mil", or one of the two-letter country codes. The use of anything else as the first component of these names is not allowed. In particular, companies generating these names must not eliminate their "com." from the string.

Again, these iSCSI names are NOT addresses. Even though they make use of DNS domain names, they are used only to specify the naming authority. An iSCSI name contains no implications of the iSCSI target or initiator's location. The use of the domain name is only a method of re-using an already ubiquitous name space.

1.2. Constructing iSCSI names using the eui. format

The iSCSI eui. naming format allows a naming authority to use IEEE

EUI-64 identifiers in constructing iSCSI names. The details of constructing EUI-64 identifiers are specified by the IEEE Registration Authority (see [EUI64]).

Example iSCSI name :

```
Type  EUI-64 identifier (ASCII-encoded hexadecimal)
+--++--------------+
|  ||              |
eui.02004567A425678D
```

2. iSCSI Alias

The iSCSI alias is a UTF-8 text string that may be used as an additional descriptive name for an initiator and target. This may not be used to identify a target or initiator during login, and does not have to follow the uniqueness or other requirements of the iSCSI name. The alias strings are communicated between the initiator and target at login, and can be displayed by a user interface on either end, helping the user tell at a glance whether the initiators and/or targets at the other end appear to be correct. The alias must NOT be used to identify, address, or authenticate initiators and targets.

The alias is a variable length string, between 0 and 255 characters, and is terminated with at least one NULL (0x00) character, as defined in [iSCSI]. No other structure is imposed upon this string.

2.1. Purpose of an Alias

Initiators and targets are uniquely identified by an iSCSI Name. These identifiers may be assigned by a hardware or software manufacturer, a service provider, or even the customer. Although these identifiers are nominally human- readable, they are likely be be assigned from a point of view different from that of the other side of the connection. For instance, a target name for a disk array may be built from the array's serial number, and some sort of internal target ID. Although this would still be human-readable and transcribable, it offers little assurance to someone at a user interface who would like to see "at-a-glance" whether this target is really the correct one.

The use of an alias helps solve that problem. An alias is simply a descriptive name that can be assigned to an initiator target, that is independent of the name, and does not have to be unique. Since it is not unique, the alias must be used in a purely informational way. It may not be used to specify a target at login, or used during authentication.

Both targets and initiators may have aliases.

2.2. Target Alias

To show the utility of an alias, here is an example using an alias

for an iSCSI target.

Imagine sitting at a desktop station that is using some iSCSI devices over a network. The user requires another iSCSI disk, and calls the storage services person (internal or external), giving any authentication information that the storage device will require for the host. The services person allocates a new target for the host, and sends the Target Name for the new target, and probably an address, back to the user. The user then adds this Target Name to the configuration file on the host, and discovers the new device.

Without an alias, a user managing an iSCSI host would click on some sort of management "show targets" button to show the targets to which the host is currently connected.

```
+--Connected-To-These-Targets----------------------
|
|  Target Name
|
|  iqn.1995-04.com.example:sn.5551212.target.450
|  iqn.1995-04.com.example:sn.5551212.target.489
|  iqn.1995-04.com.example:sn.8675309
|  iqn.2001-04.com.example.storage:tape.sys1.xyz
|  iqn.2001-04.com.example.storage.tape:sys1.xyz
|
+-------------------------------------------------
```

In the above example, the user sees a collection of iSCSI Names, but with no real description of what they are for. They will, of course, map to a system-dependent device file or drive letter, but it's not easy looking at numbers quickly to see if everything is there.

If a more intelligent target configures an alias for each target, the alias can provide a more descriptive name. This alias may be sent back to the initiator as part of the login response, or found in the iSCSI MIB. It then might be used in a display such as the following:


```
+--Connected-To-These-Targets----------------------
|
|  Alias          Target Name
|
|  Oracle 1       iqn.1995-04.com.example:sn.5551212.target.450
|  Local Disk     iqn.1995-04.com.example:sn.5551212.target.489
|  Exchange 2     iqn.1995-04.com.example:sn.8675309
|
+-------------------------------------------------
```

This would give the user a better idea of what's really there.

In general, flexible, configured aliases will probably be supported by larger storage subsystems and configurable gateways. Simpler devices will likely not keep configuration data around for things such as an alias. The TargetAlias string could be either left unsupported (not given to the initiator during login) or could be returned as whatever the "next best thing" that the target has that might better describe it. Since it does not have to be unique, it could even return SCSI inquiry string data.

Note that if a simple initiator does not wish to keep or display alias information, it can be simply ignored if seen in the login

response.

2.3. Initiator Alias

An initiator alias can be used in the same manner as a target alias. An initiator may send the alias in a login request, when it sends its iSCSI Initiator Name. The alias is not used for authentication, but may be kept with the session information for display through a management GUI or command-line interface (for a more complex subsystem or gateway), or through the iSCSI MIB.

Note that a simple target can just ignore the Initiator Alias if it has no management interface on which to display it.

Usually just the hostname would be sufficient for an initiator alias, but a custom alias could be configured for the sake of the service provider if needed. Even better would be a description of what the machine was used for, such as "Exchange Server 1", or "User Web Server".

Here's an example of a management interface showing a list of sessions on an iSCSI target network entity. For this display, the targets are using an internal target number, which is a fictional field that has purely internal significance.


```
+--Connected-To-These-Initiators-------------------
|
|  Target   Initiator Name
|
|  450      iqn.1995-04.com.example.sw:cd.12345678-OEM-456
|  451      iqn.1995-04.com.example.os:hostid.A598B45C
|  309      iqn.1995-04.com.example.sw:cd.87654321-OEM-259
|
+-------------------------------------------------
```

And with the initiator alias displayed:

```
+--Connected-To-These-Initiators-------------------
|
|  Target Alias               Initiator Name
|
|  450    Web Server 4        iqn.1995-04.com.example.sw:cd.12...
|  451    scsigw.example.com  iqn.1995-04.com.example.os:hosti...
|  309    Exchange Server     iqn.1995-04.com.example.sw:cd.87...
|
+-------------------------------------------------
```

This gives the storage administrator a better idea of who is connected to their targets. Of course, one could always do a reverse DNS lookup of the incoming IP address to determine a host name, but simpler devices really don't do well with that particular feature due to blocking problems, and it won't always work if there is a firewall or iSCSI gateway involved.

Again, these are purely informational and optional and require a management application.

Aliases are extremely easy to implement. Targets just send a TargetAlias whenever they send a TargetName. Initiators just send an InitiatorAlias whenever they send an InitiatorName. If an alias is received that does not fit, or seems invalid in any way, it is ignored.

3. iSCSI Discovery

The goal of iSCSI discovery is to allow an initiator to find the targets to which it has access, and at least one address at which

each target may be accessed. This should generally be done using as little configuration as possible. This section defines the discovery mechanism only; no attempt is made to specify central management of iSCSI devices within this document. Moreover, the iSCSI discovery mechanisms listed here only deal with target discovery and one still

needs to use the SCSI protocol for LUN discovery.

In order for an iSCSI initiator to establish an iSCSI session with an iSCSI target, the initiator needs the IP address, TCP port number and iSCSI target name information. The goal of iSCSI discovery mechanisms are to provide low overhead support for small iSCSI setups, and scalable discovery solutions for large enterprise setups. Thus, there are several methods that may be used to find targets ranging from configuring a list of targets and addresses on each initiator and doing no discovery at all, to configuring nothing on each initiator, and allowing the initiator to discover targets dynamically. The various discovery mechanisms differ in their assumptions about what information is already available to the initiators and what information needs to be still discovered.

iSCSI supports the following discovery mechanisms:

a. Static Configuration: This mechanism assumes that the IP address, TCP port and the iSCSI target name information are already available to the initiator. The initiators need to perform no discovery in this approach. The initiator uses the IP address and the TCP port information to establish a TCP connection, and it uses the iSCSI target name information to establish an iSCSI session. This discovery option is convenient for small iSCSI setups.

b. SendTargets: This mechanism assumes that the target's IP address and TCP port information are already available to the initiator. The initiator then uses this information to establish a discovery session to the Network Entity. The initiator then subsequently issues the SendTargets text command to query information about the iSCSI targets available at the particular Network Entity (IP address). SendTargets command details can be found in the iSCSI draft [iSCSI]. This discovery option is convenient for iSCSI gateways and routers.

c. Zero-Configuration: This mechanism assumes that the initiator does not have any information about the target. In this option, the initiator can either multicast discovery messages directly to the targets or it can send discovery messages to storage name servers. Currently, there are many general purpose discovery frameworks available such as Salutation[John], Jini[John], UPnP[John], SLP[RFC2608] and iSNS[iSNS]. However, with respect to iSCSI, SLP can clearly perform the needed discovery functions [iSCSI-SLP], while iSNS [iSNS] can be used to provide related management functions including notification, access management, configuration, and discovery management. iSCSI equipment that need discovery functions beyond SendTargets should at least

implement SLP, and then consider iSNS when extended discovery management capabilities are required such as in larger storage networks. It should be noted that since iSNS will support SLP, iSNS can be used to help manage the discovery information returned by SLP.

4. Security Considerations

Most security issues relating to iSCSI naming are discussed in the main iSCSI draft [iSCSI] and the iSCSI security draft [IPS-SEC].

In addition, Appendix B discusses naming and discovery issues when gateways, proxies, and firewalls are used to solve security or discovery issues in some situations where iSCSI is deployed.

iSCSI allows several different authentication methods to be used. For many of these methods, an authentication identifier is used, which may be different from the iSCSI node name of the entity being authenticated. This is discussed in more detail in Appendix C.

5. References

[SAM2] R. Weber et al, INCITS T10 Project 1157-D revision 24, "SCSI Architectural Model - 2 (SAM-2)", Section 4.7.6 "SCSI device name", September 2002.

[John] R. John, "UPnP, Jini and Salutation- A look at some popular coordination frameworks for future networked devices", http://www.cswl.com/whiteppr/tech/upnp.html", June 17, 1999.

[RFC2979] N. Freed, "Behavior of and Requirements for Internet Firewalls", RFC 2979, October 2000.

[RFC3303] P. Srisuresh et al, "Middlebox Communication Architecture and Framework", RFC 3303, August 2002.

[iSCSI] J. Satran et al, "iSCSI", Work in Progress, draft-ietf-ips-iscsi-20.txt, January 2003.

[iSNS] J. Tseng et al, "Internet Storage Name Service (iSNS)", Work in Progress, draft-ietf-ips-isns-17.txt, January 2003.

[RFC1737] K. Sollins, L. Masinter, "Functional Requirements for Uniform Resource Names", RFC 1737, December 1994.

[RFC1035] P. Mockapetris, "Domain Names - Implementation and Specification", RFC 1035, November 1987.

[EUI64] EUI - "Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority http://standards.ieee.org/regauth/oui/tutorials/EUI64.html

[RFC2396] T. Berners-Lee, R. Fielding, L. Masinter, "Uniform Resource Identifiers", RFC 2396, August 1998.

[RFC2276] K. Sollins, "Architectural Principles of URN Resolution", RFC 2276, January 1998.

[RFC2483] M. Mealling, R. Daniel, Jr., "URI Resolution Services", RFC 2483, January 1999.

[RFC2141] R. Moats, "URN Syntax", RFC 2141, May 1997.

[RFC2611] L. Daigle et al, "URN Namespace Definition Mechanisms", RFC 2611, June 1999.

[RFC2608] E. Guttman et al, "SLP Version 2", RFC 2608, June 1999.

[RFC2610] C. Perkins, E. Guttman, "DHCP Options for the Service Location Protocol", RFC 2610, June 1999.

[RFC2373] R. Hindon, S. Deering, "IP Version 6 Addressing

draft-ietf-ips-iscsi-name-disc-09
Architecture", RFC 2373, July 1998.

[RFC2732] R. Hindon, B. Carpenter, L Masinter, "Format for Literal IPv6 Addresses in URLs", RFC 2732, December 1999.

[iSCSI-SLP] M. Bakke et al, "Finding iSCSI Targets and Name Servers using SLP", Work in Progress, draft-ietf-ips-iscsi-slp-05.txt, March 2003.

[IPS-SEC] B. Aboba et al, "Securing Block Storage Protocols over IP", Work in Progress, draft-ietf-ips-security-19.txt, January 2003.

6. Authors' Addresses

Address comments to:

Kaladhar Voruganti
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
Email: kaladhar@us.ibm.com

Mark Bakke
Cisco Systems, Inc.
6450 Wedgwood Road
Maple Grove, MN 55311
Phone: +1 763 398-1054
Email: mbakke@cisco.com

Jim Hafner
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
Phone: +1 408 927-1892
Email: hafner@almaden.ibm.com

John L. Hufferd
IBM Storage Systems Group
5600 Cottle Road
San Jose, CA 95193
Phone: +1 408 256-0403
Email: hufferd@us.ibm.com

Marjorie Krueger
Hewlett-Packard Corporation
8000 Foothills Blvd
Roseville, CA 95747-5668, USA
Phone: +1 916 785-2656
Email: marjorie_krueger@hp.com

Appendix A: iSCSI Naming Notes

Some iSCSI Name Examples for Targets

- Assign to a target based on controller serial number

  iqn.2001-04.com.example:diskarray.sn.8675309

- Assign to a target based on serial number

  iqn.2001-04.com.example:diskarray.sn.8675309.oracle_db_1

Where oracle_db_1 might be a target label assigned by a user.

This would be useful for a controller that can present different logical targets to different hosts.

Obviously, any naming authority may come up with its own scheme and hierarchy for these names, and be just as valid.

A target iSCSI Name should never be assigned based on interface hardware, or other hardware that can be swapped and moved to other devices.

Some iSCSI Name Examples for Initiators

- Assign to the OS image by fully qualified host name

  iqn.2001-04.com.example.os:dns.com.customer1.host-four

Note the use of two FQDNs - that of the naming authority and also that of the host that is being named. This can cause problems, due to limitations imposed on the size of the iSCSI Name.

- Assign to the OS image by OS install serial number

  iqn.2001-04.com.example.os:newos5.12345-OEM-0067890-23456

Note that this breaks if an install CD is used more than once. Depending on the O/S vendor's philosophy, this might be a feature.

- Assign to the Raid Array by a service provider

  iqn.2001-04.com.example.myssp:users.mbakke05657

Appendix B: Interaction with Proxies and Firewalls

iSCSI has been designed to allow SCSI initiators and targets to communicate over an arbitrary network. This, making some assumptions about authentication and security, means that in theory, the whole internet could be used as one giant storage network.

However, there are many access and scaling problems that would come up when this is attempted.

1. Most iSCSI targets may only meant to be accessed by one or a few

initiators. Discovering everything would be unnecessary.

2. The initiator and target may be owned by separate entities, each with their own directory services, authentication, and other schemes. An iSCSI-aware proxy may be required to map between these things.

3. Many environments use non-routable IP addresses, such as the "10." network.

For these and other reasons, various types of firewalls [RFC2979] and proxies will be deployed for iSCSI, similar in nature to those already handling protocols such as HTTP and FTP.

B.1. Port Redirector

A port redirector is a stateless device that is not aware of iSCSI. It is used to do Network Address Translation (NAT), which can map IP addresses between routable and non-routable domains, as well as map TCP ports. While devices providing these capabilities can often filter based on IP addresses and TCP ports, they generally do not provide meaningful security, and are used instead to resolve internal network routing issues.

Since it is entirely possible that these devices are used as routers and/or aggregators between a firewall and an iSCSI initiator or target, iSCSI connections must be operable through them.

Effects on iSCSI:

- iSCSI-level data integrity checks must not include information from the TCP or IP headers, as these may be changed in between the initiator and target.

- iSCSI messages that specify a particular initiator or target, such as login requests and third party requests, should specify the initiator or target in a location-independent manner. This is accomplished using the iSCSI Name.

- When an iSCSI discovery connection is to be used through a port redirector, a target will have to be configured to return a domain name instead of an IP address in a SendTargets response, since the port redirector will not be able to map the IP address(es) returned in the iSCSI message. It is a good practice to do this anyway.

B.2. SOCKS server

A SOCKS server can be used to map TCP connections from one network domain to another. It is aware of the state of each TCP connection.

The SOCKS server provides authenticated firewall traversal for applications that are not firewall-aware. Conceptually, SOCKS is a "shim-layer" that exists between the application (i.e., iSCSI) and TCP.

To use SOCKS, the iSCSI initiator must be modified to use the encapsulation routines in the SOCKS library. The initiator the opens up a TCP connection to the SOCKS server, typically on the canonical SOCKS port 1080. A sub-negotiation then occurs, during which the initiator is either authenticated or denied the connection request. If authenticated, the SOCKS server then opens a TCP connection to the iSCSI target using addressing information sent to it by the initiator in the SOCKS shim. The SOCKS server then forwards iSCSI commands, data, and responses between the iSCSI initiator and target.

Use of the SOCKS server requires special modifications to the iSCSI initiator. No modifications are required to the iSCSI target.

As a SOCKS server can map most of the addresses and information contained within the IP and TCP headers, including sequence numbers,

its effects on iSCSI are identical to those in the port redirector.

B.3. SCSI gateway

This gateway presents logical targets (iSCSI Names) to the initiators, and maps them to SCSI targets as it chooses. The initiator sees this gateway as a real iSCSI target, and is unaware of any proxy or gateway behavior. The gateway may manufacture its own iSCSI Names, or map the iSCSI names using information provided by the physical SCSI devices. It is the responsibility of the gateway to ensure the uniqueness of any iSCSI name it manufactures. The gateway may have to account for multiple gateways having access to a single physical device. This type of gateway is used to present parallel SCSI, Fibre Channel, SSA, or other devices as iSCSI devices.

Effects on iSCSI:

- Since the initiator is unaware of any addresses beyond the gateway, the gateway's own address is for all practical purposes the real address of a target. Only the iSCSI Name needs to be passed. This is already done in iSCSI, so there are no further requirements to support SCSI gateways.

B.4. iSCSI Proxy

An iSCSI proxy is a gateway that terminates the iSCSI protocol on both sides, rather than translate between iSCSI and some other transport. The proxy functionality is aware that both sides are iSCSI, and can take advantage of optimizations, such as the preservation of data integrity checks. Since an iSCSI initiator's discovery or configuration of a set of targets makes use of address-independent iSCSI names, iSCSI does not have the same proxy addressing problems as HTTP, which includes address information into its URLs. If a proxy is to provide services to an initiator on behalf of a target, the proxy allows the initiator to discover its address for the target, and the actual target device is discovered only by the proxy. Neither the initiator nor the iSCSI protocol needs to be aware of the existence of the proxy. Note that a SCSI gateway may also provide iSCSI proxy functionality when mapping targets between two iSCSI interfaces.

Effects on iSCSI:

- Same as a SCSI gateway. The only other effect is that iSCSI must separate data integrity checking on iSCSI headers and iSCSI data, to allow the data integrity check on the data to be propagated end-to-end through the proxy.

B.5. Stateful Inspection Firewall (stealth iSCSI firewall)

The stealth model would exist as an iSCSI-aware firewall, that is invisible to the initiator, but provides capabilities found in the iSCSI proxy.

Effects on iSCSI:

- Since this is invisible, there are no additional requirements on the iSCSI protocol for this one.

This one is more difficult in some ways to implement, simply because it has to be part of a standard firewall product, rather than part of an iSCSI-type product.

Also note that this type of firewall is only effective in the outbound direction (allowing an initiator behind the firewall to

connect to an outside target), unless the iSCSI target is located in a DMZ (De-Militarized Zone) [RFC3303]. It does not provide adequate security otherwise.

Appendix C: iSCSI Names and Security Identifiers

This document has described the creation and use of iSCSI Node Names. There will be trusted environments where this is a sufficient form of identification. In these environments the iSCSI Target may have an Access Control List (ACL), which will contain a list of authorized entities that are permitted to access a restricted resource (in this case a Target Storage Controller). The iSCSI Target will then use that ACL to permit (or not) certain iSCSI Initiators to access the storage at the iSCSI Target Node. This form of ACL is used to prevent trusted initiators from making a mistake and connecting to the wrong storage controller.

It is also possible that the ACL and the iSCSI Initiator Node Name can be used in conjunction with the SCSI layer for the appropriate SCSI association of LUNs with the Initiator. The SCSI layer's use of the ACL will not be discussed further in this document.

There will be situations where the iSCSI Nodes exist in untrusted environments. That is, some iSCSI Initiator Nodes may be authorized to access an iSCSI Target Node, however, because of the untrusted environment, nodes on the network cannot be trusted to give the correct iSCSI Initiator Node Names.

In untrusted environments an additional type of identification is required to assure the target that it really knows the identity of the requesting entity.

The authentication and authorization in the iSCSI layer is independent of anything that IPSec might handle, underneath or around the TCP layer. This means that the initiator node needs to pass some type of security related identification information (e.g. userid) to a security authentication process such as SRP, CHAP, Kerberos etc. (These authentication processes will not be discussed in this document).

Upon the completion of the iSCSI security authentication, the installation knows "who" sent the request for access. The installation must then check to ensure that such a request, from the identified entity, is permitted/authorized. This form of Authorization is generally accomplished via an Access Control List (ACL) as described above. Using this authorization process, the iSCSI target will know that the entity is authorized to access the

iSCSI Target Node.

It may be possible for an installation to set a rule that the security identification information (e.g. UserID) be equal to the iSCSI Initiator Node Name. In that case, the ACL approach described above should be all the authorization that is needed.

If, however, the iSCSI Initiator Node Name is not used as the security identifier there is a need for more elaborate ACL functionality. This means that the target requires a mechanism to map the security identifier (e.g. UserID) information to the iSCSI Initiator Node Name. That is, the target must be sure that the entity requesting access is authorized to use the name, which was specified with the Login Keyword "InitiatorName=". For example, if security identifier 'Frank' is authorized to access the target via

iSCSI InitiatorName=xxxx, but 'Frank' tries to access the target via iSCSI InitiatorName=yyyy, then this login should be rejected.

On the other hand, it is possible that 'Frank' is a roaming user (or a Storage Administrator) that "owns" several different systems, and thus, could be authorized to access the target via multiple different iSCSI initiators. In this case, the ACL needs to have the names of all the initiators through which 'Frank' can access the target.

There may be other more elaborate ACL approaches, which can also be deployed to provide the installation/user with even more security with flexibility.

The above discussion is trying to inform the reader that, not only is there a need for access control dealing with iSCSI Initiator Node Names, but in certain iSCSI environments there might also be a need for other complementary security identifiers.

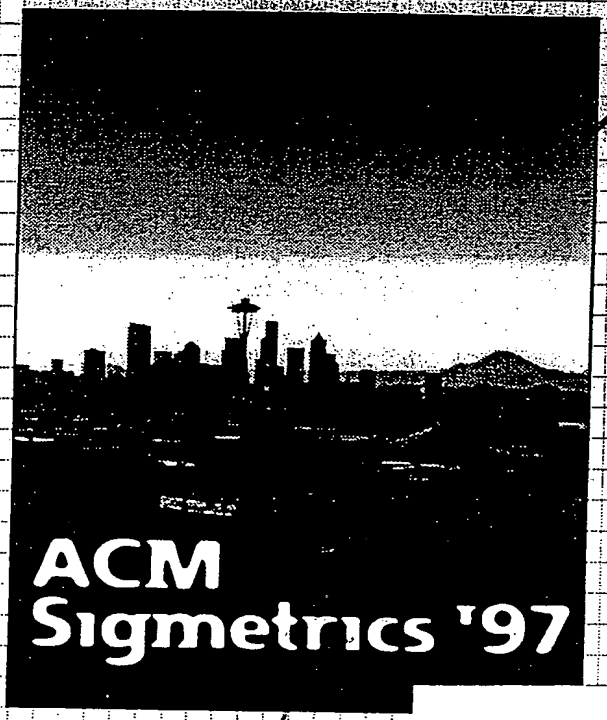
7. Full Copyright Notice

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.

PERFORMANCE EVALUATION REVIEW

SPECIAL ISSUE VOLUME 25 no.1 JUNE 1997



THE 1997 ACM SIGMETRICS INTERNATIONAL CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS

PROCEEDINGS

June 15 - 18 1997 Seattle Washington U.S.A.

File Server Scaling with Network-Attached Secure Disks

Garth A. Gibson†, David F. Nagle*, Khalil Amiri*, Fay W. Chang†, Eugene M. Feinberg*, Howard Gobioff†, Chen Lee†, Berend Ozceri*, Erik Riedel*, David Rochberg†, Jim Zelenka†

*Department of Electrical and Computer Engineering
†School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890
garth+nasd@cs.cmu.edu
http://www.cs.cmu.edu/Web/Groups/NASD/

Abstract

By providing direct data transfer between storage and client, network-attached storage devices have the potential to improve scalability for existing distributed file systems (by removing the server as a bottleneck) and bandwidth for new parallel and distributed file systems (through network striping and more efficient data paths). Together, these advantages influence a large enough fraction of the storage market to make commodity network-attached storage feasible. Realizing the technology's full potential requires careful consideration across a wide range of file system, networking and security issues. This paper contrasts two network-attached storage architectures—(1) Networked SCSI disks (NetSCSI) are network-attached storage devices with minimal changes from the familiar SCSI interface, while (2) Network-Attached Secure Disks (NASD) are drives that support independent client access to drive object services. To estimate the potential performance benefits of these architectures, we develop an analytic model and perform trace-driven replay experiments based on AFS and NFS traces. Our results suggest that NetSCSI can reduce file server load during a burst of NFS or AFS activity by about 30%. With the NASD architecture, server load (during burst activity) can be reduced by a factor of up to five for AFS and up to ten for NFS.

1 Introduction

Users are increasingly using distributed file systems to access data across local area networks; personal computers with hundred-plus MIPS processors are becoming increasingly affordable; and the sustained bandwidth of magnetic disk storage is expected to exceed 30 MB/s by the end of the decade. These trends place a pressing need on distributed file system architectures to provide clients with efficient, scalable, high-bandwidth access to stored data. This paper discusses a powerful approach to fulfilling this need. Network-attached storage provides high bandwidth by directly attaching storage to the network, avoiding file server store-and-forward operations and allowing data transfers to be striped over storage and switched-network links.

The principal contribution of this paper is to demonstrate the potential of network-attached storage devices for penetrating the markets defined by existing distributed file system clients, specifically the Network File System (NFS) and Andrew File System (AFS) distributed file system protocols. Our results suggest that network-attached storage devices can improve overall distributed file system cost-effectiveness by offloading disk access, storage management and network transfer and greatly reducing the amount of server work per byte accessed.

We begin by charting the range of network-attached storage devices that enable scalable, high-bandwidth storage systems. Specifically, we present a taxonomy of network-attached storage — server-attached disks (SAD), networked SCSI (NetSCSI) and network-attached secure disks (NASD) — and discuss the distributed file system functions offloaded to storage and the security models supportable by each.

With this taxonomy in place, we examine traces of requests on NFS and AFS file servers, measure the operation costs of commonly used SAD implementations of these file servers and develop a simple model of the change in manager costs for NFS and AFS in NetSCSI and NASD environments. Evaluating the impact on file server load analytically and in trace-driven replay experiments, we find that NASD promises much more efficient file server offloading in comparison to the simpler NetSCSI. With this potential benefit for existing distributed file server markets, we conclude that it is worthwhile to engage in detailed NASD implementation studies to demonstrate the efficiency, throughput and response time of distributed file systems using network-attached storage devices.

In Section 2, we discuss related work. Section 3 presents our taxonomy of network-attached storage architectures. In Section 4, we describe the NFS and AFS traces used in our analysis and replay experiments and report our measurements of the cost of each server operation in CPU cycles. Section 5 develops an analytic model to estimate the potential scaling offered by server-offloading in NetSCSI and NASD based on the collected traces and the measured costs of server operations. The trace-driven replay experiment and the results are the subject of Section 6. Finally, Section 7 presents our conclusions and discusses future directions.

This research was sponsored by DARPA/ITO through ARPA Order D306 under contract N00174-96-0002 and in part by an ONR graduate fellowship. The project team is indebted to generous contributions from the member companies of the Parallel Data Consortium: Hewlett-Packard, Symbios Logic Inc., Data General, Compaq, IBM Corporation, EMC Corporation, Seagate Technology, and Storage Technology Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of any supporting organization or the U.S. Government.

SIGMETRICS '97 Seattle, WA, USA

2 Related Work

Distributed file systems provide remote access to shared file storage in a networked environment [Sandberg85, Howard88, Minshall94]. A principal measure of a distributed file system's cost is the computational power required from the servers to provide adequate performance for each client's work [Howard88, Nelson88]. While microprocessor performance is increasing dramatically and raw computational power would not normally be a concern, the work done by a file server is data- and interrupt-intensive and, with the poorer locality typical of operating systems, faster microprocessors will provide much less benefit than their cycle time trends promise [Ousterhout91, Anderson91, Chen93].

Typically, distributed file systems employ client caching to reduce this server load. For example, AFS clients use local disk to cache a subset of the global system's files. While client caching is essential for high performance, increasing file sizes, computation sizes, and workgroup sharing are all inducing more misses per cache block [Ousterhout85, Baker91]. At the same time, increased client cache sizes are making these misses more bursty.

When the post-client-cache server load is still too large, it can either be distributed over multiple servers or satisfied by a custom-designed high-end file server. Multiple-server distributed file systems attempt to balance load by partitioning the namespace and replicating static, commonly used files. This replication and partitioning is too often ad-hoc, leading to the "hotspot" problem familiar in multiple-disk mainframe systems [Kim86] and requiring frequent user-directed load balancing. Not surprisingly, custom-designed high-end file servers more reliably provide good performance, but can be an expensive solution [Hitz90, Drapeau94].

Experience with disk arrays suggests another solution. If data is striped over multiple independent disks of an array, then a high-concurrency workload will be balanced with high probability as long as individual accesses are small relative to the unit of interleaving [Linvy87, Patterson88, Chen90]. Similarly, striping file storage across multiple servers provides parallel transfer of large files and balancing of high concurrency workloads [Hartman93]; striping of metadata promises further load-balancing [Dahlin95].

Scalability prohibits the use of a single shared-media network; however, with the emergence of switched network fabrics based on high-speed point-to-point links, striped storage can scale bandwidth independent of other traffic in the same fabric [Arnould89, Siu95, Boden95]. Unfortunately, current implementations of Internet protocols demand significant processing power to deliver high bandwidth — we observe as much as 80% of a 233 MHz DEC Alpha consumed by UDP/IP receiving 135 Mbps over 155 Mbps ATM (even with adaptor support for packet reassembly). Improving this bandwidth depends on interface board designs [Steenkiste94, Cooper90], integrated layer processing for network protocols [Clark89], direct application access to the network interface [vonEiken92, Maeda93], copy avoiding buffering schemes [Druschel93, Brustoloni96], and routing support for high-performance best-effort traffic [Ma96, Traw95]. Perhaps most importantly, the protocol stacks resulting from these research efforts must be deployed widely. This deployment is critical because the comparable storage protocols, SCSI, and soon, Fibre Channel, provide cost-effective hardware implementations routinely included in client machines. For comparison, a 175 MHz DEC Alpha consumes less than 5% of its processing power fetching 100 Mbps from a 160 Mbps SCSI channel via the UNIX raw disk interface.

To exploit the economics of large systems resulting from the cobbling together of many client purchases, the xFS file system distributes code, metadata and data over all clients, eliminating the need for a centralized storage system [Dahlin95]. This scheme naturally matches increasing client performance with increasing server performance. Instead of reducing the server workload, however, it takes the required computational power from another, frequently idle, client. Complementing the advantages of filesystems such as xFS, the network-attached storage architectures presented in this paper significantly reduce the demand for server computation and eliminate file server machines from the storage data path, reducing the coupling between overall file system integrity and the security of individual client machines.

As distributed file system technology has improved, so have the storage technologies employed by these systems. Storage density increases, long a predictable 25% per year, have risen to 60% increases per year during the 90s. Data rates, which were constrained by storage interface definitions until the mid-80s, have increased by about 40% per year in the 90s [Grochowski96]. The acceptance, in all but the lowest cost market, of SCSI, whose interface exports the abstraction of a linear array of fixed-size blocks provided by an embedded controller [ANSI86], catalyzed rapid deployment of technology advances, resulting in an extremely competitive storage market.

The level of indirection introduced by SCSI has also led to transparent improvements in storage performance such as RAID; transparent failure recovery; real-time geometry-sensitive scheduling; buffer caching; read-ahead and write-behind; compression; dynamic mapping; and representation migration [Patterson88, Gibson92, Massiglia94, StorageTek94, Wilkes95, Ruemmler91, Varma95]. However, in order to overcome the speed, addressability and connectivity limitations of current SCSI implementations [Sachs94, ANSI95], the industry is turning to high-speed packetized interconnects such as Fibre Channel at up to 1 Gbps [Benner96]. The disk drive industry anticipates the marginal cost for on-disk Fibre Channel interfaces, relative to the common single-ended SCSI interface in use today, to be comparable to the marginal cost for high-performance differential SCSI (a difference similar to the cost of today's Ethernet adapters) while their host adapter costs are expected to be comparable to high-performance SCSI adapters [Anderson95].

The idea of simple, disk-like network-based storage servers whose functions are employed by higher-level distributed file systems, has been around for a long time [Birrel80, Katz92]. The Mass Storage System Reference Model (MSSRM), an early architecture for hierarchical storage subsystems, has advocated the separation of control and data paths for almost a decade [Miller88, IEEE94]. Using a high-bandwidth network that supports direct transfers for the data path is a natural consequence [Kronenberg86, Drapeau94, Long94, Lee95, Menascé96, VanMeter96]. The MSSRM has been implemented in the High Performance Storage System (HPSS) [Watson95] and augmented with socket-level striping of file transfers [Berdahl95, Wiltzius95], over the multiple network interfaces found on mainframes and supercomputers.[1]

[1]Following Van Meter's [VanMeter96] definition of network-attached peripherals, we consider only networks that are shared with general local area network traffic and not single-vendor systems whose interconnects are fast, isolated local area networks [Horst95, IEEE92].

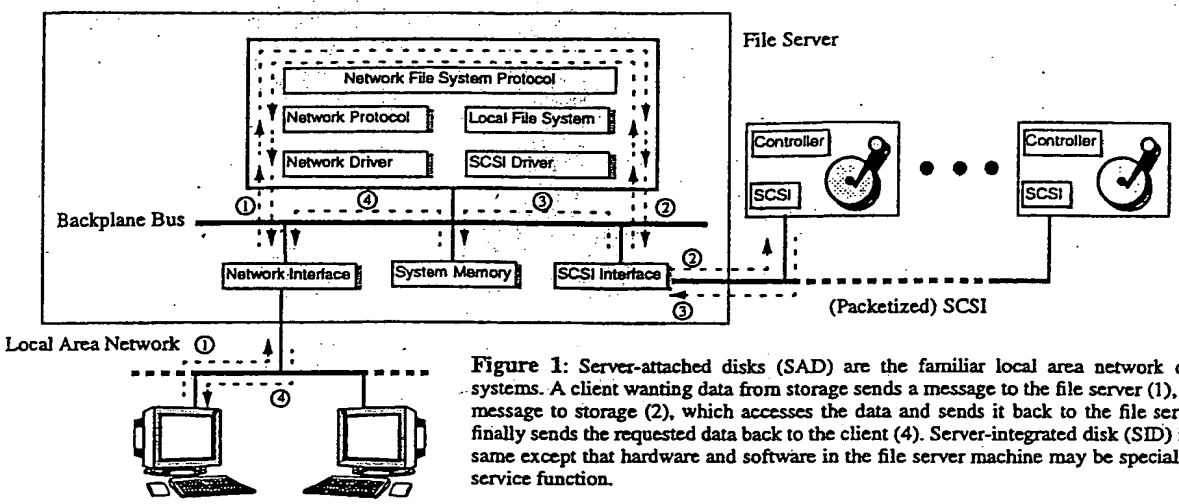


Figure 1: Server-attached disks (SAD) are the familiar local area network distributed file systems. A client wanting data from storage sends a message to the file server (1), which sends a message to storage (2), which accesses the data and sends it back to the file server (3), which finally sends the requested data back to the client (4). Server-integrated disk (SID) is logically the same except that hardware and software in the file server machine may be specialized to the file service function.

Striping data across multiple storage servers with independent ports into a scalable local area network has been advocated as a means of obtaining scalable storage bandwidth [Hartman93]. If the storage servers of this architecture are network-attached devices, rather than dedicated machines between the network and storage, efficiency is further improved by avoiding store-and-forward delays through the server.

Our notion of network-attached storage is consistent with these projects. However, our analysis focuses on the evolution of commodity storage devices rather than niche-market, very high-end systems, and on the interaction of network-attached storage with common distributed file systems. Because all prior work views the network-based storage as a function provided by an additional computer, instead of the storage devices itself, cost-effectiveness has never been within reach. Our goal is to chart the way network-attached storage is likely to appear in storage products, estimate its scalability implications, and characterize the security and file system design issues in its implementation.

3 Taxonomy of Network-Attached Storage

Simply attaching storage to a network underspecifies network-attached storage's role in distributed file systems' architectures. In the following subsections, we present a taxonomy for the functional composition of network-attached storage. Case 0, the base case, is the familiar local area network with storage privately connected to file server machines — we call this *server-attached disks*. Case 1 represents a wide variety of current products, *server-integrated disks*, that specialize hardware and software into an integrated file server product. In Case 2, the obvious network-attached disk design, *network SCSI*, minimizes modifications to the drive command interface, hardware and software. Finally, Case 3, *network-attached secure disks*, leverages the rapidly increasing processor capability of disk-embedded controllers to restructure the drive command interface.

3.1 Case 0: Server-Attached Disks (SAD)

This is the system familiar to office and campus local area networks as illustrated in Figure 1. Clients and servers share a network and storage is attached directly to general-purpose workstations that provide distributed file services.

3.2 Case 1: Server Integrated Disks (SID)

Since file server machines often do little other than service distributed file system requests, it makes sense to construct specialized systems that perform only file system functions and not general-purpose computation. This architecture is not fundamentally different from SAD. Data must still move through the server machine before it reaches the network, but specialized servers can move this data more efficiently than general-purpose machines. Since high performance distributed file service benefits the productivity of most users, this architecture occupies an important market niche [Hitz90, Hitz94]. However, this approach binds storage to a particular distributed file system, its semantics, and its performance characteristics. For example, most server-integrated disks provide NFS file service, whose inherent performance has long been criticized [Howard88]. Furthermore, this approach is undesirable because it does not enable distributed file system and storage technology to evolve independently. Server striping, for instance, is not easily supported by any of the currently popular distributed file systems. Binding the storage interface to a particular distributed file system hampers the integration of such new features [Birrell80].

3.3 Case 2: Network SCSI (NetSCSI)

The other end of the spectrum is to retain as much as possible of SCSI, the current dominant mid- and high-level storage device protocol. This is the natural evolution path for storage devices; Seagate's Barracuda FC is already providing packetized SCSI through Fibre Channel network ports to directly attached hosts [Seagate96]. NetSCSI is a network-attached storage architecture that makes minimal changes to the hardware and software of SCSI disks. File manager software translates client requests into commands to disks, but rather than returning data to the file manager to be forwarded, the NetSCSI disks send data directly to clients, similar to the support for third-party transfers already supported by SCSI [Drapeau94]. The efficient data transfer engines typical of fast drives ensure that the drive's sustained bandwidth is available to clients. Further, by eliminating the file manager from the data path, its workload per active client decreases. However, the use of third-party transfer changes the drive's role in the overall security of a distributed file system. While it is not unusual for distributed file systems to employ a security protocol between clients and

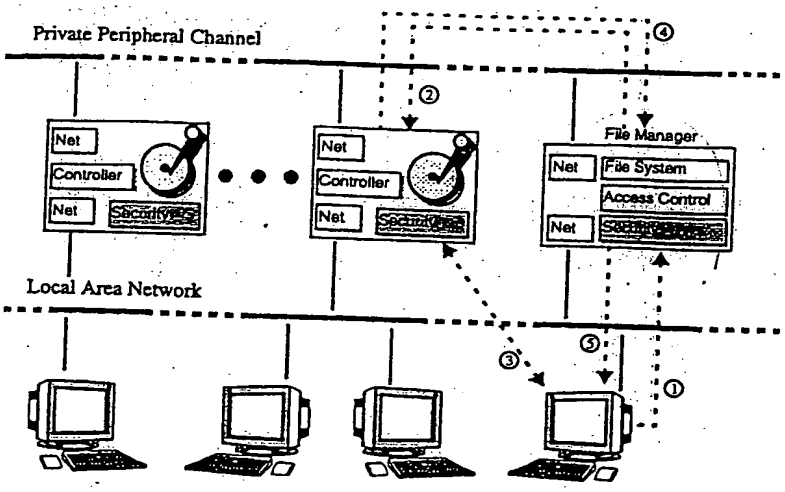


Figure 2: Network SCSI (NetSCSI) is a network-attached disk architecture designed for minimal changes to the disk's command interface. However, because the network port on these disks may be connected to a hostile, broader network, preserving the integrity of on-disk file system structure requires a second port to a private (file manager-owned) network or cryptographic support for a virtual private channel to the file manager. If a client wants data from a NetSCSI disk, it sends a message (1) to the distributed file system's file manager which processes the request in the usual way, sending a message over the private network to the NetSCSI disk (2). The disk accesses data, transfers it directly to the client (3), and sends its completion status to the file manager over the private network (4). Finally, the file manager completes the request with a status message to the client (5).

servers (e.g. Kerberos authentication), disk drives do not yet participate in this protocol.

We identify four levels of security within the NetSCSI model: (1) accident-avoidance with a second private network between file manager and disk, both locked in a physically secure room; (2) data transfer authentication with clients and drives equipped with a strong cryptographic hash function; (3) data transfer privacy with both clients and drives using encryption and; (4) secure key management with a secure coprocessor.

Figure 2 shows the simplest security enhancement to NetSCSI: a second network port on each disk. Since SCSI disks execute every command they receive without an explicit authorization check, without a second port even well-meaning clients can generate erroneous commands and accidentally damage parts of the file system. The drive's second network port provides protection from accidents while allowing SCSI command interpreters to continue following their normal execution model. This is the architecture employed in the SIOF and HPSS projects at LLNL [Wiltzius95, Watson95]. Assuming that file manager and NetSCSI disks are locked in a secure room, this mechanism is acceptable for the trusted network security model of NFS [Sandberg85].

Because file data still travels over the potentially hostile general network, NetSCSI disks are likely to demand greater security than simple accident avoidance. Cryptographic protocols can strengthen the security of NetSCSI. A strong cryptographic hash function, such as SHA [NIST94], computed at the drive and at the client would allow data transfer authentication (i.e., the correct data was received only if the sender and receiver compute the same hash on the data).

For some applications, data transfer authentication is insufficient, and communication privacy is required. To provide privacy, a NetSCSI drive must be able to encrypt and decrypt data. NetSCSI drives can use cryptographic protocols to construct private virtual channels over the untrusted network. However, since keys will be stored in devices vulnerable to physical attack, the servers must still be stored in physically secure environments. If we go one step further and equip NetSCSI disks with secure coprocessors [Tygar95], then keys can be protected and all data can be encrypted when outside the secure coprocessor, allowing the disks to be used in a variety of physically open environments. There are now a variety of secure coprocessors [NIST94a, Weingart87, White87, National96] available, some of which promise cryptographic accelerators sufficient to support single-disk bandwidths.

3.4 Case 3: Network-attached Secure Disks (NASD)

With network-attached secure disks, we relax the constraint of minimal change from the existing SCSI interface and implementation. Instead we focus on selecting a command interface that reduces the number of client-storage interactions that must be relayed through the file manager, offloading more of the file manager's work without integrating file system policy into the disk.

Common data-intensive operations, such as reads and writes, go straight to the disk, while less-common ones, including namespace and access control manipulations, go to the file manager. As opposed to NetSCSI, where a significant part of the processing for security is performed on the file manager, NASD drives perform most of the processing to enforce the security policy. Specifically, the cryptographic functions and the enforcement of manager decisions are implemented at the drive, while policy decisions are made in the file manager. Because clients directly request access to data in their files, a NASD drive must have sufficient metadata to map and authorize the request to disk sectors. Authorization, in the form of a time-limited capability applicable to the file's map and contents, should be provided by the file manager to protect higher-level file systems' control over storage access policy. The storage mapping metadata, however, could be provided dynamically [VanMeter96a] by the file manager or could be maintained by the drive. While the latter approach asks distributed file system authors to surrender detailed control over the layout of the files they create, it enables smart drives to better exploit detailed knowledge of their own resources to optimize data layout, read-ahead, and cache management [deJonge93, Patterson95, Golding95]. This is precisely the type of value-added opportunity that nimble storage vendors can exploit for market and customer advantage. With mapping metadata at the drive controlling the layout of files, a NASD drive exports a namespace of file-like objects. Because control of naming is more appropriate to the higher-level file system, pathnames are not understood at the drive, and pathname resolution is split between the file manager and client. While a single drive object will suffice to represent a simple client file, multiple objects may be logically linked by the file system into one client file. Such an interface provides support for banks of

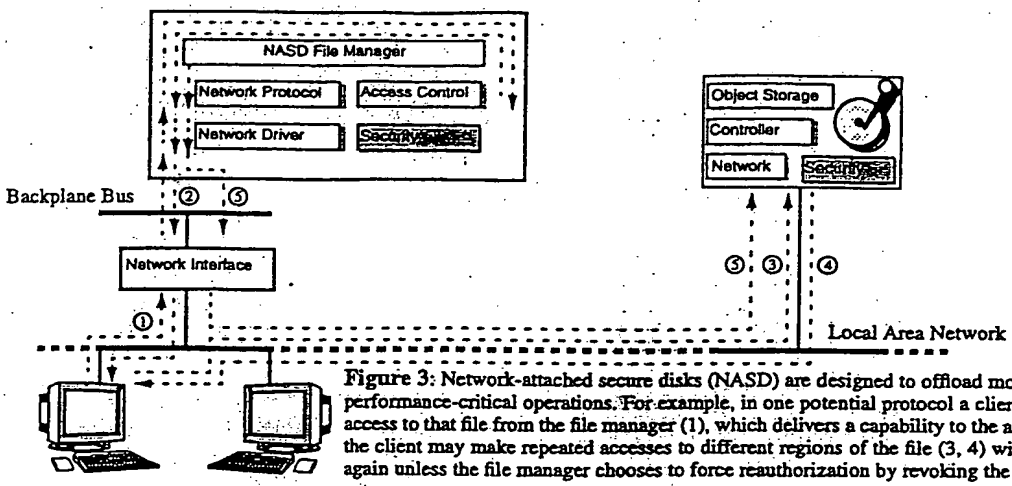


Figure 3: Network-attached secure disks (NASD) are designed to offload more of the file system's simple and performance-critical operations. For example, in one potential protocol a client, prior to reading a file, requests access to that file from the file manager (1), which delivers a capability to the authorized client (2). So equipped, the client may make repeated accesses to different regions of the file (3, 4) without contacting the file manager again unless the file manager chooses to force reauthorization by revoking the capability (5).

striped files [Hartman93], Macintosh-style resource forks, or logically-contiguous chunks of complex files [deJong93].

As an example of a possible NASD access sequence, consider a file read operation depicted in Figure 3. Before issuing its first read of a file, the client authenticates itself with the file manager and requests access to the file. If access is granted, the client receives the network location of the NASD drive containing the object and a time-limited capability to access the object and for establishing a secure communications channel with the drive. After this point, the client may directly request access to data on NASD drives, using the appropriate capability [Gobioff96].

In addition to offloading file read operations from the distributed file manager, later sections will show that NASD should also offload file writes and attributes reads to the drive. High-level file system policies, such as access control and cache consistency, however, remain the purview of the file manager. These policies are enforced by NASD drives according to the capabilities controlled by the file manager.

3.5 Summary

This taxonomy, summarized in Table 1, splits into two classes — SAD and SID offer a specific distributed file system while NetSCSI and NASD offer enhanced storage interfaces. The difference between SID and NASD merits further consideration. Many of the optimizations we propose for NASD, such as shortened data paths and specialized protocol processing, can also be implemented in a SID architecture. However, SID binds storage to a particular distributed file system, requires higher-level (or multiple-SID) file management to offer network striped files and, by not evolving the drive interface, inhibits the independent development of drive technology. For the rest of this paper, we focus on SAD, NetSCSI, and NASD and present a coarse-grained estimate of the potential benefit of network-attached storage. The results suggest that by exploiting the processing power available in next generation storage devices, computation required from the file manager machines can be dramatically reduced, enabling the per-byte cost of distributed file service to be reduced.

	Case 0	Case 1	Case 2	Case 3
FM per byte	X	X		
FM per operation			X	
FM on open/close				X
specialization		X	X	X

Table 1. Comparison of network-attached storage architectures. SAD and SID require the file manager (file server) to handle each byte of data, but SID allows specialization of the hardware and software to file service. NetSCSI allows direct transfers to clients, but requires file manager interaction on each operation to manage metadata.

4 Analysis of File System Workload

To develop an understanding of performance parameters critical to network-attached storage, we performed a series of measurements to (1) characterize the behavior and cost of AFS and NFS distributed file server functionality; and (2) identify and subset busy periods during which server load is limiting.

4.1 Trace Data

Our data is taken from NFS and AFS file system traces summarized in Table 2. The NFS trace [Dahlin94] records the activity of an Auspex file server supporting 231 client machines over a one week period at the University of California at Berkeley[2]. The AFS trace records the activity of our laboratory's Sparcstation 20 AFS server supporting 250 client machines over a one month period[3]

	NFS trace	AFS trace
Number of client machines	231	250
Total number of requests	6,676,479	1,615,540
Read data transferred (GB)	8.1	2.9
Write data transferred (GB)	2.0	1.6
Trace period	9/20/93-9/24/93 40 hours	9/9/96-10/3/96 435 hours[3]

Table 2. Description of the traces used in the experiments. The NFS trace was collected in a study performed at the University of California at Berkeley. The AFS trace was collected by logging requests at the AFS file server in our laboratory.

[2]Some attribute reads were removed from the NFS trace by the Berkeley researchers based on a heuristic for eliminating excessive cache consistency traffic. Because this change is pessimistic to our proposed architecture, we choose to continue to use these traces, already familiar to the community, rather than collect new traces.

[3]The trace covers three periods of activity - 9/9-10, 9/13-15, and 9/20-10/3.

Table 3(a) - NFS Trace Operations

Trace Record	NFS Operations	Description	Percent	Quantity (millions)	% of Cycles
AttrRead	getattr	Get metadata information	42.5	2.84	11.8
AttrWrite	setattr	Update metadata information	0.3	0.02	0.3
BlockRead	read	Get data from server	20.4	1.36	31.6
BlockWrite	write	Send data to server	4.2	0.28	19.3
DirRead	lookup, readdir	Convert filename to filehandle, get directory entries	31.4	2.10	35.5
DirReadWrite	create, mkdir, rename, etc.	Create new files/directories, rename, etc.	1.0	0.07	1.0
DeleteWrite	unlink, rmdir	Remove file/directory	0.2	0.01	0.4

Table 3(b) - NFS Cost Measurements

Data Size (bytes)	Read Cycles (thousands)	Write Cycles (thousands)
1	54	117
1K	61	—
2K	68	—
4K	78	148
8,000	100	199

Table 3(c)

Operation	Cycles (thousands)
getattr	33
setattr	64
lookup	50
readdir (1 entry)	63
readdir (40 entries)	105

Table 3(d)

Operation	Cycles (thousands)
create	81
unlink (last link)	135

Table 3: Distribution and average costs of NFS operations. Cycle counts were taken on an ATOMized DEC 3000/400 (133 MHz, 64 MB of memory, Digital UNIX 3.2c) kernel, including NFSv3 server functionality. ATOM overhead was calculated and removed. The server's caches were warmed, and trials that produced misses in the buffer cache were discarded. The write and create operations were measured using a RAM-based file system.

Both the NFS and AFS traces document each client request with an arrival timestamp, a unique client host id, and an indication of the request type. The AFS trace records the exact type of primitive AFS file system request and also includes a response timestamp. The NFS trace only records the general class of the issued request which leaves some ambiguity in determining exactly which primitive NFS requests were issued (e.g., a request recorded as a directory read may have been either a lookup or readdir request).

The original NFS trace is dominated by overnight backup activity. Since users are mostly insensitive to backup performance, and this is not a major concern of this study, we exclude this activity by only including requests timestamped Monday through Friday between 9am and 5pm in our data set. The AFS trace does not include any backup activity because AFS backups are handled by a separate task on the file server machine.

4.2 Cost of NFS and AFS Operations

Our trace data captures the types and relative frequencies of client requests but does not include the amount of CPU work performed by the file server in handling each request. To estimate this cost, we measured NFS and AFS server code paths on Digital Equipment Alpha workstations. Specifically, we used the ATOM binary annotation tool [Srivastava94] and the Alpha's on-chip cycles counters to identify the code paths traversed and measure the work required for each type of primitive file system operation.

To minimize measurement overhead and improve accuracy, cost measurements were taken in two steps. First, we used ATOM to annotate the entry and exit points of each procedure and issued specific requests, producing a dynamic call graph for each primitive operation. Then we re-annotated the server routines, starting at packet arrival and ending at response packet dispatch, limiting annotation to the critical components of each operation's code path. For each operation, file system requests were repeatedly applied to the selectively annotated server, generating traces that recorded code-path execution times. Measurements were repeated for a range of request sizes where appropriate and all the measurements are summarized in Table 3(b,c,d) and Table 4(b,c,d).

4.3 Relative Importance of NFS and AFS Operations

Table 3(a) and Table 4(a) report the frequency distribution of various server operations for the traces. Each table describes the types of primitive operations and reports their frequencies and the total number of occurrences in the trace. This data shows that attribute read requests (AttrRead, FetchStatus, BulkStatus) are the most frequently executed operations. While frequency statistics emphasize attribute operations, the cycle count data indicate that data movement can place a significantly larger per request burden on the server CPU.

To assess the relative importance of various primitive operations in the total workload applied to a file server, we estimate the total amount of work performed by a server per request type during the execution of each trace. Specifically, we estimate the total server workload per operation type by multiplying the per-type count of occurrences by the measured average per-type cycle count. Since the NFS trace groups certain operation types together, as indicated by Column 2 of Table 3(a) we use a representative member of each group to perform our NFS calculations. The relative importance of operation types can be deduced from the percentage of the server load attributable to each type, as shown in the last column of Table 3(a) and Table 4(a). These calculations show that data-moving operations contribute 51% of the NFS workload and 36% of the AFS workload. Because these fractions are far short of 100%, the performance gained by directly moving data between clients and disks may be limited [Drapeau94]. As the next subsection shows, this limits the benefit of NetSCSI for offloading file manager workload and motivates the design of a NASD drive interface.

Table 4(a) - AFS Trace Operations

AFS Operation	Description	Percent	Quantity (thousands)	% of Cycles
FetchStatus	Get metadata information	65.1	1052.2	39.3
BulkStatus	Perform a group of FetchStatus operations	5.8	93.4	10.9
StoreStatus	Update metadata information	2.5	40.4	2.2
FetchData	Get data from server	13.9	224.0	27.9
StoreData	Send data to server	3.8	61.5	8.5
CreateFile	Create a new file	1.7	27.0	2.4
Rename	Rename a file/directory	0.6	10.4	0.9
RemoveFile	Remove a file	1.5	25.0	2.4
Others	ACL manipulation, symbolic links, directory creation/deletion, lock management, etc.	5.0	81.6	5.4

Table 4(b) - AFS Cost Measurements

	Cycles according to Size of Operation (thousands)										
Operation	0	1	512	1K	2K	4K	8K	16K	32K	64K	1M
FetchData	—	179	192	191	204	270	330	439	788	1,544	—
StoreData	259	—	291	303	363	371	410	578	750	1,242	16,752
RemoveFile	—	331	396	396	410	411	412	414	429	452	1,053

Table 4(c)

BulkStatus Size (directory entries)	Cycles (thousands)
1	151
3	178
10	324
20	578
25	662

Table 4(d)

Operation	Cycles (thousands)
FetchStatus	128
StoreStatus	189
CreateFile	307
Rename	285
Others	227

Table 4: Distribution and average costs of AFS operations. Cycle counts were taken on a DEC 3000/500 (150MHz, 128 MB of memory, Digital UNIX 3.2c) running an ATOMized AFS version 3.4 server. ATOM tracing overhead was negligible compared to other system-level effects on the server. The server's caches were warmed and trials that produced misses in the local file system cache were discarded. The number of cycles for "Others" was estimated as the average of the four size-independent operations that were measured individually (FetchStatus, StoreStatus, CreateFile, Rename).

4.4 Busy Client-Minutes

A distributed file system scales if an increase in aggregate client demand, and the corresponding increase in storage capacity and bandwidth, does not result in a decrease in client-observed performance. In a previous study [Riedel96], we examined the correlation between hourly averages of client response times, network round-trip times and server load. Users may be satisfied with their response times when servers are idle, but experience periods of dramatically longer response times which correlate with periods of high server load. Since client dissatisfaction is strongly determined by prolonged periods of considerably higher than average response time, this study focuses on server performance during such bursts of high load. For such a burst to have client impact, it must persist for a sufficiently long time. In this paper we have chosen to examine load during one minute intervals — long enough for interactive users to identify a slowdown, but not so long that poor performance during bursts is hidden by overall averages. Our previous study also observed that periods of high server load may exhibit a different distribution of request types — data movement is more prevalent. In order to capture the distribution of operations during these critical bursty periods, we restrict the rest of our analysis to

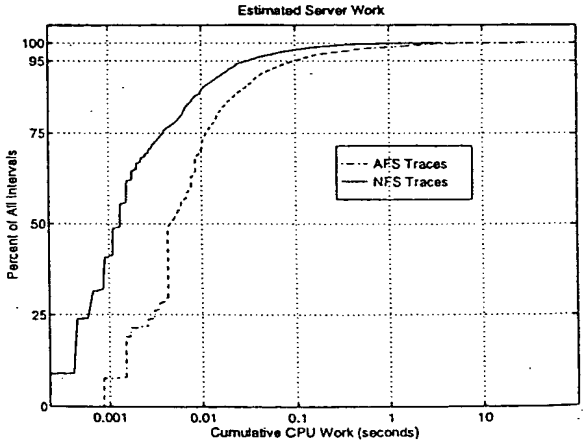


Figure 4: Cumulative distribution of estimated server work for NFS and AFS intervals. The graph shows that 98% of NFS client-minutes and 95% of AFS client-minutes require less than 0.1 seconds of estimated server work.

	NFS		AFS	
	Number	% of total	Number	% of total
Busy client-minutes	4,636	2	2,809	5
Client machines	135	58	78	31
Requests	3,730,031	56	1,199,419	74
Read data (GB)	4.8	59	2.8	96
Write data (GB)	1.7	84	1.3	84

Table 5: Statistics for the top 2% of NFS client-minutes, and top 5% of AFS client-minutes, as measured by estimated work.

the busiest one-minute intervals as measured by the amount of work detailed in Section 4.3.

Based on this metric and the data in Figure 4, we chose to restrict analysis to *client-minutes* (single minutes of a single client's activity) that consume more than 0.1 seconds of server CPU (top 2% of NFS and top 5% of AFS client-minutes). Table 5 summarizes these busy client-minutes.

5 Analytic Model

When a distributed file system is ported to NetSCSI or NASD environments, the disposition of client requests is adjusted according to the goals described in Section 3. The principal benefit we expect for an existing file system such as NFS or AFS is a more cost-effective scaling of throughput by a reduction in the file manager load. In this section, we develop a simple estimate of this scaling. Following the work estimates of Section 4.3, where total file manager work is estimated as the sum of operation costs weighted by the frequency of each operation, we derive estimates of the NetSCSI and NASD file manager work done by NFS and AFS operations by approximating these costs with SAD operations which accomplish similar amounts of work, as reported in Table 6. These estimates are only coarse approximations, but provide a reasonable estimate of the potential benefit of network-attached storage over SAD in terms of file manager scaling.

In the NetSCSI model, the only change from SAD is that the read/write datapath avoids the file manager. However, each read or write request must still be authorized and translated to NetSCSI block addresses. As we see in Table 7, this severely limits the scalability of NetSCSI even though we optimistically model the manager cost of read or write as a simple attribute read in SAD. Specifically, this model estimates file manager work with NetSCSI to be at least two-thirds and three-quarters as much as with SAD during busy NFS and AFS client-minutes.

In our NASD model, all read operations, including attribute and directory reads, are sent directly to the NASD drive. We further assume that NFS clients in NASD systems replace directory lookup operations with NASD (directory) object reads and execute the lookup locally. The data in file writes are also sent directly to the NASD drive. However, in order to support AFS consistency semantics, data writes generate an additional request to the file manager. This request, which would allow the AFS file manager to perform the appropriate consistency maintenance (e.g. breaking callbacks), is estimated to require the same work as an attribute read request. NFS, which has a weaker consistency model, does not require this additional request. For attribute and directory writes, we assume that clients must send their requests to the file manager. To estimate the file manager's pre-authorization and capability setup work prior to any access, we introduced a NASD open request which we emulate with an attribute read operation. Since NASD capabilities are valid for a limited time (twenty-four hours in this model), unless revoked by a change in access rights (an operation that is extremely rare in our traces), transforming the traces in this way adds one additional operation when a file is first referenced on a given day. Finally, remove operations, whose deallocation work is done by the NASD drive, require file manager work comparable to the removal of an empty file.

For AFS, Table 7 shows that NASD systems may reduce file manager workload during busy client-minutes by a factor of two over NetSCSI systems and a factor of three over SAD systems. For NFS, where directory and attribute reads dominate the workload, file managers using NASD drives may benefit from a factor of fourteen decrease in file manager load over SAD systems.

6 Replay Experiment

The analytic model neglects several factors. Particularly concerning is its inability to account for system-level activity (e.g. page faults, scheduler activity, thread overhead, queueing effects) that could significantly impact the behavior and performance of NetSCSI and NASD systems. Given our goal of justifying further implementation studies, we chose to explore system overheads and interactions by replaying the traces, modified according to Table 6 to coarsely model the work of a NetSCSI or NASD server, against existing SAD implementations. This experiment allows us to measure expected file manager load under SAD, NetSCSI and NASD,

NFS trace	SAD	NetSCSI	NASD
AttrRead	getattr	getattr	—
AttrWrite	setattr	setattr	setattr
BlockRead	read(size)	getattr	—
BlockWrite	write(size)	getattr	—
DirRead	lookup	lookup	—
DirRW	create	create	create
DeleteWrite	remove(size)	remove(size)	remove(0 byte)
NasdOpen	—	—	getattr

AFS trace	SAD	NetSCSI	NASD
FetchStatus	FetchStatus	FetchStatus	—
StoreStatus	StoreStatus	StoreStatus	StoreStatus
FetchData	FetchData(size)	FetchStatus	—
StoreData	StoreData(size)	FetchStatus	FetchStatus
RemoveFile	RemoveFile(size)	RemoveFile(size)	RemoveFile(0 byte)
BulkStatus	BulkStatus	BulkStatus	—
NasdOpen	—	—	FetchStatus

Table 6. Description of what the operations in the filesystem traces translate to in the SAD, NetSCSI and NASD models. The tables list the operations as recorded in the trace and the corresponding RPC request issued by a client during replay for each of SAD, NetSCSI and NASD and used in the analytic calculations to estimate server load in each model. The operations not listed for AFS are the same across SAD, NetSCSI and NASD. The last row in each table corresponds to the NASD open operation, which we added to the traces.

NFS Operation	Count in top 2% by work (thousands)	SAD		NetSCSI		NASD	
		Cycles (billions)	%	Cycles (billions)	%*	Cycles (billions)	%*
Attr Read	792.7	26.4	11.8%	26.4	11.8%	0.0	0.0%
Attr Write	10.0	0.6	0.3%	0.6	0.3%	0.6	0.3%
Block Read	803.2	70.4	31.6%	26.8	12.0%	0.0	0.0%
Block Write	228.4	43.2	19.4%	7.6	3.4%	0.0	0.0%
Dir Read	1577.2	79.1	35.5%	79.1	35.5%	0.0	0.0%
Dir RW	28.7	2.3	1.0%	2.3	1.0%	2.3	1.0%
Delete Write	7.0	0.9	0.4%	0.9	0.4%	0.9	0.4%
Open	95.2	0.0	0.0%	0.0	0.0%	12.2	5.5%
Total	3542.4	223.1	100.0%	143.9	64.5%	16.1	7.2%

AFS Operation	Count in top 5% by work (thousands)	SAD		NetSCSI		NASD	
		Cycles (billions)	%	Cycles (billions)	%*	Cycles (billions)	%*
FetchStatus	770.5	98.6	37.9%	98.6	37.9%	0.0	0.0%
BulkStatus	91.3	36.6	14.1%	36.6	14.1%	0.0	0.0%
StoreStatus	16.2	3.1	1.2%	3.1	1.2%	3.1	1.2%
FetchData	193.7	83.7	32.1%	24.8	9.5%	0.0	0.0%
StoreData	23.1	15.1	5.8%	3.0	1.1%	3.0	1.1%
CreateFile	12.1	3.7	1.4%	3.7	1.4%	3.7	1.4%
Rename	6.4	1.8	0.7%	1.8	0.7%	1.8	0.7%
RemoveFile	14.6	4.8	1.9%	4.8	1.9%	4.8	1.9%
Others	57.3	13.0	5.0%	13.0	5.0%	13.0	5.0%
Open	480.8	0.0	0.0%	0.0	0.0%	61.5	23.6%
Total	1665.9	260.5	100.0%	189.4	72.7%	90.9	34.9%

Table 7: Estimated work performed by the NFS and AFS file managers to handle requests issued during busy client-minutes. This table reports the estimates of our analytic model comparing the relative scalability of file managers in SAD, NetSCSI, and NASD environments. "%" in the NetSCSI and NASD columns represents the percentage difference between each particular NetSCSI or NASD operation cycle count and the SAD total cycle count.

capturing system-level activities not accounted for in the analytic model and more accurately estimating the increase in scalability possible with NetSCSI- and NASD-based systems.

6.1 Experiment

The replay environment, as illustrated in Figure 5, is composed of a single file manager (i.e. an NFS or AFS server) and several host workstations. We refer to these host workstations, used to replay (modified) trace requests to the file manager, as *replay hosts*. Each replay host merges several client-minute traces which it then replays using an open-loop request-issue model where multiple threads replay each of the requests according to the issue timestamps. When the total load applied is well under the server's capability, the timing of operations approximates the original traces and the replay completes in about one minute. However, as the number of client-minutes grows, the work required of the server exceeds its capability and responses may be delayed so long that all client threads are blocked when a timestamp requires a request to be replayed. When such deadlines are missed often, the system degrades to a closed-loop experiment and the runtime can be significantly longer than one minute. During replay, file manager CPU load is measured by recording time spent in the kernel idle loop, and subtracting this from the total duration of the replay.

To measure file server load as a function of increasing client demand, we varied the number of client-minutes replayed simultaneously. To replay m client-minutes, we randomly select m client-minutes from the pool of busy client-minutes described in Section 4.4. As indicated by the long tails of the distributions in Figure 4, different randomly selected sets of m client-minutes may have

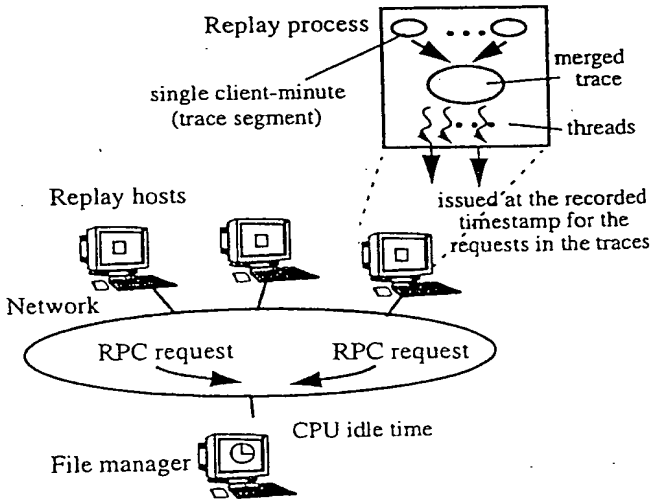


Figure 5: Setup of the trace-driven replay experiments. Multi-threaded processes on each replay host submit requests from a set of client-minutes to the file server emulating the expected traffic in the case of SAD, NetSCSI, and NASD.

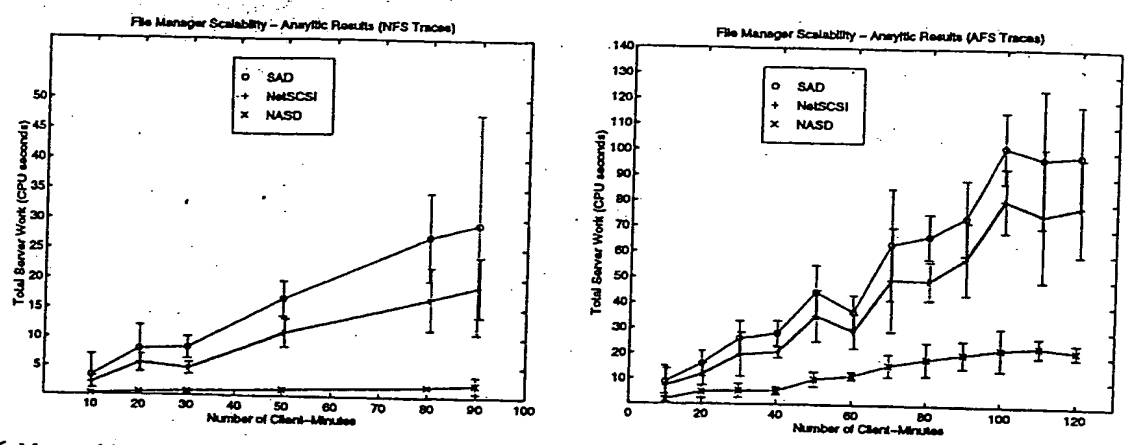


Figure 6: Mean and 90% confidence intervals according to the analytic model described in Section 5 applied to the five randomly selected samples of m client-minutes constructed for the replay experiment.

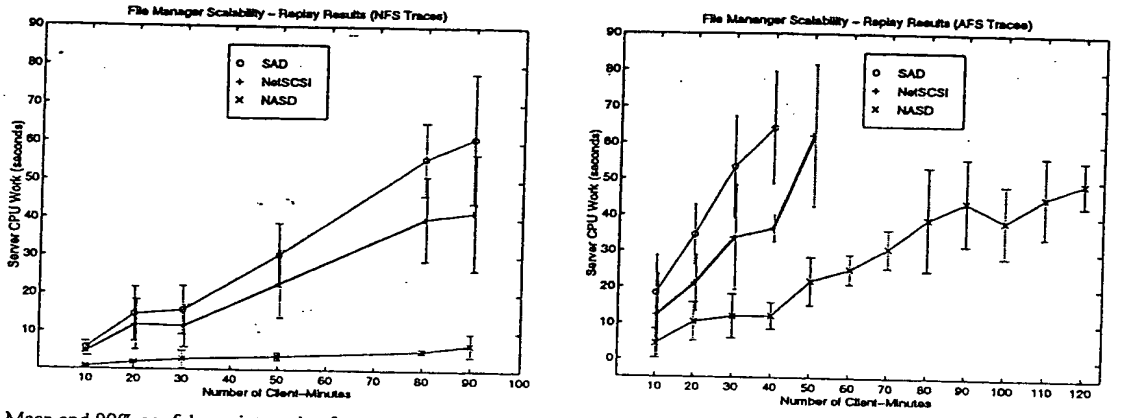


Figure 7. Mean and 90% confidence intervals of measured load for NFS and AFS replay experiments for five samples of m client-minutes.

The file manager was a DEC 3000/500 (150 MHz, 128 MB of memory, Digital UNIX 3.2g) with five fast wide differential SCSI busses, each with four HP C2247 disks. To balance I/O, we stripe data across the twenty disks using a 64KB stripe unit. The replay hosts were ten DEC AlphaStation 255, 3000/400, and 3000/600 workstations interconnected by a switched OC-3 ATM network (NFS replay used up to eight additional machines connected via Ethernet).

Prior to replaying a set of client-minutes, we build a filesystem which allows the clients to access files touched in those client-minutes (the original file system hierarchy was not collected with the traces). We create the correct number of files, sizing each file heuristically as the largest offset accessed in the ace. In NFS replay, each replay host was responsible for c (=5) client-minutes. Therefore, the number of replay hosts, h, varied with the number of -minutes replayed. In AFS replay, the client-minutes were always evenly divided amongst h (=10) replay hosts.

widely varying load. Therefore, we construct p (=5) samples for each set of m client-minutes, and report the mean and 90% confidence intervals. For comparison, Figure 6 reports the mean and 90% confidence intervals in estimated file manager work according to the analytic model of Section 5 applied to the client-minutes selected for replay.

6.2 Results

Comparing Figure 6 and Figure 7, we see that replay experiences significantly more CPU work — work that was overlooked by the analytic model. At 90 client-minutes, the analytic NFS/NASD model predicts less than 30 CPU seconds while the replay model consumes over 60 CPU seconds. In spite of these differences, the replay results and the analytic data display a strong correlation in the relative performance of SAD, NetSCSI and NASD. For example, at 90 NFS client-minutes, both show a 40% difference in load between SAD and NetSCSI, and a 90% difference between SAD and NASD, with similar correspondence in the AFS case. The similarities between the results of Section 5 and 6 suggest that, provided implementations on NetSCSI and NASD have operation costs similar to those in Table 6, NetSCSI provides limited benefit to existing distributed file systems (a factor of about 1.5 improvement). In contrast, NASD promises substantially lower file manager costs per client (factors of up to ten for NFS and up to five for AFS).

From Figure 7, it appears that each data point required less than 60 seconds of file manager CPU, the amount available in these one minute replay experiments. However, when CPU saturation of the manager slows the generation of trace events, it causes the replay to run longer than 60 seconds. For NFS, replay overrun occurs with 80 or more client-minutes in SAD and NetSCSI and does not occur in NASD. For AFS, this overrun occurs with 25 or more client-minutes in SAD, 50 or more client-minutes in NetSCSI and 120 or more client-minutes in NASD. AFS suffers

more from this effect because its file manager is user-level with only one kernel thread — the entire file manager blocks on every disk access (NFS is in-kernel and the file manager has sixteen threads at its disposal).

6.3 Cache Effects

A limitation of the results in Figure 7 relates to our handling of file manager cache state. The use of samples constructed from random, busy one-minute intervals makes it difficult to determine what constitutes a realistic initial state for the data and metadata caches. Because a cache miss induces more file manager work than a cache hit, biases which increase misses also increase work. Further, because the file manager work in SAD, NetSCSI and NASD differs, with far fewer cache accesses done by NetSCSI and NASD, it is reasonable to expect that SAD file manager work is over-estimated more by excess misses than the other cases.

For this reason we should have run all workloads with warm caches, biasing in favor of SAD file managers. Unfortunately, our ability to control cache contents carefully was best when using cold caches.[4] Therefore, to bound the bias against SAD, we ran a simple experiment. For NFS's 10 and 20 client-minute workloads, the entire set of data accessed during those client-minutes fits in the file manager's buffer cache. This allowed us to perform the NFS replay with an initially cold data cache, then repeat the same replay without flushing the contents of the cache (thus starting from an optimally-warmed cache, containing all the data which will be accessed in the second run). In this case, the CPU load on the file manager in SAD decreased by 10-18%, which we take as the upper bound on the effect of warm versus cold data caches on the CPU load of a SAD file manager. To restate, Figure 7 may falsely penalize SAD performance by up to 18% because of cold data caches during replay.

7 Conclusion and Future Directions

Network-attached storage, by enabling direct transfers between client and storage, can substantially increase distributed file system scalability while simultaneously enabling striped storage to satisfy the bursty, high-bandwidth demands of the increasingly high-performance clients populating local area networks. This promises benefits in a wide enough range of storage markets and makes commodity network-attached storage feasible.

In this paper we have presented a simple classification of storage architectures for distributed file systems with four models. The traditional, server-attached disk (SAD) model is our base case. Server-integrated (SID) disk systems, including specialized NFS server products, are architecturally identical, but have hardware and software designed specifically for file service. We do not emphasize this model because it binds storage products to a particular choice of distributed file system.

The remaining two storage models exploit the potential of network-attached storage. Network SCSI (NetSCSI) drives are very similar to current SCSI disks in that all file requests go through the file manager, but the resulting data transfers go directly between client and drive. This may reduce file manager workload during busy periods by about 30%. Different security models can be provided using NetSCSI depending on the cryptographic support provided in the drive.

Network-attached secure disks (NASD) support storage semantics at a level between that of block-level protocols like SCSI and distributed file systems like NFS and AFS. The partitioning of file system functionality between NASD drive and file manager is optimized to reduce file manager load while maintaining system flexibility. To operate securely in the face of this partition, NASD drives rely on cryptographic support for security and authorization. Our studies show that, by offloading data read and write and attribute and directory read operations, distributed file system server load during busy periods may be reduced by a factor of fourteen (NFS) and three (AFS) in the analytic model, and up to ten (NFS) and five (AFS) in the replay experiments.

Our analysis focuses on describing the distinct methods of organizing storage architecture and estimating the potential improvement each promises for existing distributed file systems. With the positive results given here, our future directions are clear. We plan to demonstrate that distributed file systems can be implemented around network-attached storage, preserving powerful security models and yielding considerable scalability and client performance advantages. Along this path, many open questions remain. Our NASD model, in particular, expects a disk drive to be capable of computation not normally associated with cost-sensitive commodity peripherals; drive micro-architectures and software structures must be developed and demonstrated.

Further, NASD's out-of-datapath file manager does not naturally provide the server caching found in traditional systems which store-and-forward data through the server. We must evaluate the penalty of distributing the caches among storage. This penalty may be mitigated if storage objects are striped over drives because striping inherently eliminates hotspots [Livny87]. On the other hand, server caching is significantly less important to performance than client caching and becomes less important still with cooperative caching in idle clients [Dahlin94, Feeley95] and aggressive prefetching by clients [Patterson95, Cao95].

Finally, in the NASD models presented, we assume that clients "open" files by contacting the distributed file system server one file at a time to set up the state needed for direct transfers to and from storage and allow the file manager to handle consistency. A clear improvement, similar to the effect of client caching in AFS, might be provided by pre-authorization or group-authorization schemes.

8 Acknowledgments

We would like to thank Michael Dahlin from Berkeley for providing us their NFS traces and helping us understand some of the details. We thank Doug Tygar for his comments. We would also like to thank all of the members of the Parallel Data Lab who allowed us to trace their AFS accesses over the course of several months and provided much helpful feedback on the ideas and experiments presented here. Finally, we thank the anonymous reviewers for their helpful comments.

9 Bibliography

[Anderson91]Anderson, T.E. et al., "The Interaction of Architecture and Operating System Design," 4th ASPLOS, Sept. 1991.

[Anderson95]Anderson, D. (Seagate Technology), Personal communication, 1995.

[ANSI86]ANSI, "Small Computer System Interface (SCSI) Specification", ANSI X3.131-1986, 1986.

[4]Even here our control was incomplete; NFS used a cold data cache, but a warm metadata cache. AFS uses both cold data and metadata caches.

[ANSI95]ANSI, "SCSI-3 Fast-20 Parallel Interface", X3T10/1047D Working Group, Revision 6.

[Arnould89]Arnould, E.A. et al., "The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers", 3rd ASPLOS, April 1989, pp. 205-216.

[Baker91]Baker, M.G. et al., "Measurements of a Distributed File System", 13th SOSP, Oct. 1991, pp. 198-212.

[Benner96]Benner, A.F., "Fibre Channel: Gigabit Communications and I/O for Computer Networks", McGraw Hill, New York, 1996.

[Berdahl95]Berdahl, L., Draft of "Parallel Transport Protocol Proposal", Lawrence Livermore National Labs, January 3, 1995.

[Birrell80]Birrell, A.D. and Needham, R.M., "A Universal File Server", IEEE Transactions on Software Engineering SE-6,5, Sept. 1980.

[Boden95]Boden, N.J. et al, "Myrinet: A Gigabit-per-Second Local Area Network", IEEE Micro, Feb. 1995.

[Brustoloni96] Brustoloni, G. and Steenkiste, P., "Effects of Buffering Semantics on I/O Performance," 2nd OSDI, Oct. 1996.

[Cao95]Cao, P. et al., "A Study of Integrated Prefetching and Caching Strategies," SIGMETICS 95, May 1995.

[Chen90]Chen, P.M. et al., "An evaluation of Redundant Arrays of Disks using an Amdahl 5890," SIGMETRICS 90, 1990.

[Chen93]Chen, J.B. and Bershad, B., "The Impact of Operating System Structure on Memory System Performance," 14th SOSP, Dec. 1993, pp. 120-133.

[Clark89]Clark, D.D. et al., "An Analysis of TCP Processing Overhead," IEEE Communications 27,6 (June 89), pp. 23-36.

[Cooper90]Cooper, E., et al., "Host Interface Design for ATM LANs", 16th Conference on Local Computer Networks, Oct. 1991, pp. 247-258.

[Dahlin94]Dahlin, M. et al., "Cooperative Caching: Using Remote Client Memory to Improve File System Performance," First OSDI, pp. 267-280, Nov. 1994.

[Dahlin95]Dahlin, M.D. et al., "A Quantitative Analysis of Cache Policies for Scalable Network File Systems", 15th SOSP, Dec. 1995.

[deJong93]deJonge, W., Kaashoek, M.F. and Hsieh, W.C., "The Logical Disk: A New Approach to Improving File Systems," 14th SOSP, Dec. 1993.

[Drapeau94]Drapeau, A.L. et al., "RAID-II: A High-Bandwidth Network File Server", 21st ISCA, 1994, pp. 234-244.

[Druschel93]Druschel, P. and Peterson, L.L., "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility", 14th SOSP, Dec. 1993, pp. 189-202.

[Feeley95]Feeley, M. J. et al., "Implementing global memory management in a workstation cluster," 15th SOSP, Dec. 1995.

[Gibson92]Gibson, G., "Redundant Disk Arrays: Reliable, Parallel Secondary Storage," MIT Press, 1992.

[Gobioff96]Gobioff, H. et al., "Security for Network-Attached Storage Devices," CMU-CS-96-179, 1996.

[Golding95]Golding, R., et al., "Attribute-managed storage," Workshop on Modeling and Specification of I/O, San Antonio, TX, Oct. 1995.

[Grochowski96]Grochowski, E.G., Hoyt, R.F., "Future Trends in Hard Disk Drives," IEEE Transactions on Magnetics 32, 3 (May 1996), pp. 1850-1854.

[Hartman93]Hartman, J.H. and Ousterhout, J.K., "The Zebra Striped Network File System", 14th SOSP, Dec. 1993.

[Hitz90]Hitz, D. et al., "Using UNIX as One Component of a Lightweight Distributed Kernel for Multiprocessor File Servers", Winter 1990 USENIX, pp. 285-295.

[Hitz94]Hitz, D., Lau, J. and Malcolm, M. "File Systems Design for an NFS File Server Appliance", Winter 1994 USENIX, Jan. 1994.

[Horst95]Horst, R.W., "TNet: A Reliable System Area Network", IEEE Micro, Feb. 1995.

[Howard88]Howard, J.H. et al., "Scale and Performance in a Distributed File System", ACM TOCS 6, 1, Feb. 1988, pp. 51-81.

[IEEE92]IEEE, "Scalable Coherent Interconnect", Standard 1596-1992, 1992.

[IEEE94]IEEE P1244. "Reference Model for Open Storage Systems Interconnection-Mass Storage System Reference Model Version 5", Sept. 1995.

[Katz92]Katz, R.H., "High-Performance Network- and Channel-Attached Storage", Proceedings of the IEEE 80,8, Aug. 1992.

[Kim86]Kim, M.Y., "Synchronized disk interleaving", IEEE Transactions on Computers C-35, 11, Nov. 1986.

[Kronenberg86]Kronenberg, N.P. et al., "VAXClusters: A closely-coupled distributed system", ACM TOCS 4,2, May 1986, pp. 130-146.

[Lee95]Lee, E.K., "Highly-Available, Scalable Network Storage", Spring 1995 COMPCON, Mar. 1995.

[Livny87]Livny, M., "Multi-disk management algorithms", SIGMETRICS 87, May 1987.

[Long94]Long, D.D.E., Montague, B.R., and Cabrera, L., "Swift/RAID: A Distributed RAID System," Computing Systems 7,3, Summer 1994.

[Ma96]Ma, Q., Steenkiste, P., and Zhang, H., "Routing High-Bandwidth Traffic in Max-Min Fair Share Networks," SIGCOMM 96, Aug. 1996.

[Maeda93]Maeda, C., and Bershad, B., "Protocol Service Decomposition for High-Performance Networking", 14th SOSP, Dec. 1993, pp. 244-255.

[Massiglia94]Massiglia, P., ed., "The RAIDbook", RAID Advisory Board, 1994.

[Menascé96]Menascé, D.A., et al., "An Analytic Model of Hierarchical Mass Storage Systems with Network-Attached Storage Devices," SIGMETRICS 96, May 1996.

[Miller88]Miller, S.W., "A Reference Model for Mass Storage Systems", Advances in Computers 27, 1988, pp. 157-210.

[Minshall94]Minshall, G., Major, D., and Powell, K., "An Overview of the NetWare Operating System", Winter 1994 USENIX, 1994.

[National96]National Semiconductor. "The PersonaCard 100 Data Security Card," http://www.ipsecure.com/htm/persona.html.

[Nelson88]Nelson, M.N., Welch, B.B. and Ousterhout, J.K., "Caching in the Sprite Network File System", ACM TOCS 6,1, Feb. 1988, pp. 134-154.

[NIST94]National Institute of Standards and Technology, "Digital Signature Standard." NIST FIPS Pub 186.

[NIST94a]National Institute of Standards and Technology, "Security Requirements for Cryptographic Modules", NIST FIPS 140-1.

[Ousterhout85]Ousterhout, J.K. et al., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System", 10th SOSP, Dec. 1985.

[Ousterhout91]Ousterhout, J.K., "Why Aren't Operating Systems Getting Faster As Fast As Hardware?", Summer 1991 USENIX, June 1991, pp. 247-256.

[Patterson88]Patterson, D.A., Gibson, G. and Katz, R.H., "A Case for Redundant Arrays of Inexpensive Disks (RAID)", SIGMOD 88, June 1988, pp. 109-116.

[Patterson95]Patterson, R.H. et al., "Informed Prefetching and Caching", 15th SOSP, 1995.

[Rambus92]Rambus Inc., "Rambus Architectural Overview", 1992. http://www.rambus.com.

[Riedel96]Riedel, E. and Gibson, G. "Understanding Customer Dissatisfaction With Underutilized Distributed File Servers", 5th Goddard Conference on Mass Storage Systems and Technologies, College Park, MD, Sept. 1996.

[Ruemmler91]Ruemmler, C. and Wilkes, J., "Disk Shuffling", Hewlett-Packard Laboratories Concurrent Systems Project Tech Report HPL-CSP-91-30.

[Sachs94]Sachs, M.W., Leff, A., and Sevigny, D., "LAN and I/O Convergence: A Survey of the Issues", IEEE Computer, Dec. 1994. pp. 24-32.

[Sandberg85]Sandberg, R. et al., "Design and Implementation of the Sun Network Filesystem", Summer 1985 USENIX, June 1985, pp. 119-130.

[Seagate96]Seagate Corporation, "Barracuda Family Product Brief (ST19171)", 1996.

[Siu95]Siu, K.-Y. and Jain, R. "A brief overview of ATM: Protocol layers, LAN emulation and traffic management", ACM SIGCOMM, Vol 25,2, Dec. 1994, pp. 69-79.

[Srivastava94]Srivastava, A., and Eustace, A., "ATOM: A system for building customized program analysis tools", WRL Technical Report TN-41, 1994.

[StorageTek94]Storage Technology Corporation, "Iceberg 9200 Storage System: Introduction", STK Part Number 307406101, Storage Technology Corporation, 1994.

[Steenkiste94]Steenkiste, P., "A Systematic Approach to Host Interface Design for High-Speed Networks", IEEE Computer, Mar. 1994.

[Traw95]Traw, C.B.S. and Smith, J.M., "Striping Within the Network Subsystem", IEEE Network, Jul./Aug. 1995.

[Tygar95]Tygar, J.D., and Yee, B.S., "Secure Coprocessors in Electronic Commerce Applications," 1995 USENIX Electronic Commerce Workshop, 1995, New York.

[VanMeter96]Van Meter, R., "A Brief Survey Of Current Work on Network Attached Peripherals (Extended Abstract)", Operating Systems Review 30,1, Jan. 1996.

[VanMeter96a]Van Meter, R., Holtz, S., and Finn G., "Derived Virtual Devices: A Secure Distributed File System Mechanism", 5th Goddard Conference on Mass Storage Systems and Technologies", College Park, MD, Sept. 1996.

[Varma95]Varma, A. and Jacobson, Q., "Destage Algorithms for Disk Arrays with Non-volatile Caches", 22nd ISCA, 1995.

[vonEicken92]von Eicken, T. et al., "Active Messages: A Mechanism for Integrated Communication and Computation", 19th ISCA, May 1992, pp. 256-266.

[Watson95]Watson, R.W., and Coyne, R.A., "The Parallel I/O Architecture of the High-Performance Storage System (HPSS)," 14th IEEE Symposium on Mass Storage Systems, Sept. 1995, pp. 27-44.

[Weingart87]Weingart, S.H., "Physical Security of the μABYSS System", IEEE Computer Society Conference on Security and Privacy, 1987, pp. 52-58.

[White87]White, S.R. and Comerford, L., "ABYSS: A Trusted Architecture for Software Protection", IEEE Computer Society Conference on Security and Privacy, 1987, pp. 38-51.

[Wilkes95]Wilkes, J. et al., "The HP AutoRAID Hierarchical Storage System", 15th SOSP, Dec. 1995.

[Wiltzius95]Wiltzius, D. et al., "Network-attached peripherals for HPSS/SIOF", http://www.llnl.gov/liv_comp/siof/siof_nap.

[Ging 87] Gingell, R.A., Moran, J.P., and Shannon, W.A., "Virtual Memory Architecture in SunOS," *Proceedings of the Summer 1987 USENIX Technical Conference*, Jun. 1987, pp. 81–94.

[Hend 90] Hendricks, D., "A FileSystem for Software Development," *Proceedings of the Summer 1990 USENIX Technical Conference*, Jun. 1990, pp. 333-340.

[Klei 86] Kleiman, S.R., "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *Proceedings of the Summer 1986 USENIX Technical Conference*, Jun. 1986, pp. 238–247.

[Kowa 78] Kowalski, T., "FSCK—The UNIX System Check Program," Bell Laboratory, Murray Hill, N.J. 07974, Mar. 1978.

[Krid 83] Kridle, R., and McKusick, M., "Performance Effects of Disk Subsystem Choices for VAX Systems Running 4.2BSD UNIX," Technical Report No. 8, Computer Systems Research Group, Dept. of EECS, University of California at Berkeley, CA, 1983.

[Leff 89] Leffler, S.J., McKusick, M.K., Karels, M.J., and Quarterman, J.S., *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, Reading, MA, 1989.

[McKu 84] McKusick, M.K., Joy, W.N., Leffler, S.J., and Fabry, R.S., "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, vol. 2, (Aug. 1984), pp. 181–197.

[McKu 85] McKusick, M.K., Karels, M., and Leffler, S.J., "Performance Improvements and Functional Enhancements in 4.3BSD," *Proceedings of the Summer 1985 USENIX Conference*, Jun. 1985, pp. 519–531.

[McKu 90] McKusick, M.K., Karels, M.K., and Bostic, K., "A Pageable Memory Based Filesystem," *Proceedings of the Summer 1990 USENIX Technical Conference*, Jun. 1990.

[Nadk 92] Nadkarni, A.V., "The Processor File System in UNIX SVR4.2," *Proceedings of the 1992 USENIX Workshop on File Systems*, May 1992, pp. 131–132.

[Ohta 90] Ohta, M. and Tezuka, H., "A Fast /tmp File System by Delay Mount Option," *Proceedings of the Summer 1990 USENIX Conference*, Jun. 1990, pp. 145–149.

[Oust 85] Ousterhout, J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M. and Thompson, J.G., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the Tenth Symposium on Operating System Principles*, Dec. 1985, pp. 15–24.

[Rifk 86] Rifkin, A.P., Forbes, M.P., Hamilton, R.L., Sabrio, M., Shah, S., and Yueh, K., "RFS Architectural Overview," *Proceedings of the Summer 1986 USENIX Technical Conference*, Jun. 1986, pp. 248–259.

[Salu 94] Salus, P.H., *A Quarter Century of UNIX*, Addison-Wesley, Reading, MA, 1994.

[Saty 81] Satyanarayan, M., "A Study of File Sizes and Functional Lifetimes," *Proceedings of the Eighth Symposium on Operating Systems Principles*, 1981, pp. 96–108.

[Snyd 90] Snyder, P., "tmpfs: A Virtual Memory File System," *Proceedings of the Autumn 1990 European UNIX Users' Group Conference*, Oct. 1990, pp. 241–248.

[Thom 78] Thompson, K., "UNIX Implementation," *The Bell System Technical Journal*, Jul.-Aug. 1978, Vol. 57, No. 6, Part 2, pp. 1931–1946.

10

Distributed File Systems

10.1 Introduction

Since the 1970s, the ability to connect computers to each other on a network has revolutionized the computer industry. The increase in network connectivity has fueled a desire to share files between different computers. The early efforts in this direction were restricted to copying entire files from one machine to another, such as the *UNIX-to-UNIX copy (uucp)* program [Nowi 90] and *File Transfer Protocol (ftp)* [Post 85]. Such solutions, however, do not come close to fulfilling the vision of being able to access files on remote machines as though they were on local disks.

The mid-1980s saw the emergence of several distributed file systems that allow transparent access to remote files over a network. These include the *Network File System (NFS)* from Sun Microsystems [Sand 85a], the *Remote File Sharing system (RFS)* from AT&T [Rifk 86], and the *Andrew File System (AFS)* from Carnegie-Mellon University [Saty 85]. All three are sharply different in their design goals, architecture, and semantics, even though they try to solve the same fundamental problem. Today, RFS is available on many System V-based systems. NFS has gained much wider acceptance and is available on numerous UNIX and non-UNIX systems. AFS development has passed on to Transarc Corporation, where it has evolved into the *Distributed File System (DFS)* component of Open Software Foundation's *Distributed Computing Environment (DCE)*.

This chapter begins by discussing the characteristics of distributed file systems. It then describes the design and implementation of each of the above-mentioned file systems and examines their strengths and weaknesses.

10.2 General Characteristics of Distributed File Systems

A conventional, centralized file system allows multiple users on a single system to share access to files stored locally on the machine. A distributed file system extends the sharing to users on different machines interconnected by a communication network. Distributed file systems are implemented using a client-server model. The client is a machine that accesses a file, while a server is one that stores the file and allows clients to access it. Some systems may require clients and servers to be distinct machines, while others may allow a single machine to act as both client and server.

It is important to note the distinction between distributed file systems and distributed operating systems [Tann 85]. A distributed operating system, such as *V* [Cher 88] or *Amoeba* [Tann 90], is one that looks to its users like a centralized operating system, but runs simultaneously on multiple machines. It may provide a file system that is shared by all its host machines. A distributed file system, however, is a software layer that manages communication between conventional operating systems and file systems. It is integrated with the operating systems of the host machines and provides a distributed file access service to systems with centralized kernels.

There are several important properties of distributed file systems [Levy 90]. Each file system may have some or all of these properties. This gives us a basis to evaluate and compare different architectures.

- **Network transparency**—Clients should be able to access remote files using the same operations that apply to local files.
- **Location transparency**—The name of a file should not reveal its location in the network.
- **Location independence**—The name of the file should not change when its physical location changes.
- **User mobility**—Users should be able to access shared files from any node in the network.
- **Fault tolerance**—The system should continue to function after failure of a single component (a server or a network segment). It may, however, degrade in performance or make part of the file system unavailable.
- **Scalability**—The system should scale well as its load increases. Also, it should be possible to grow the system incrementally by adding components.
- **File mobility**—It should be possible to move files from one physical location to another in a running system.

10.2.1 Design Considerations

There are several important issues to consider in designing a distributed file system. These involve tradeoffs in functionality, semantics, and performance. We can compare different file systems according to how they deal with these issues:

- **Name space** — Some distributed file systems provide a uniform name space, such that each client uses the same pathname to access a given file. Others allow each client to customize its name space by mounting shared subtrees to arbitrary directories in the file hierarchy. Both methods have some appeal.

- **Stateful or stateless operation** — A stateful server is one that retains information about client operations between requests and uses this state information to service subsequent requests correctly. Requests such as *open* and *seek* are inherently stateful, since someone must remember which files a client has opened, as well as the seek offset in each open file. In a stateless system, each request is self-contained, and the server maintains no persistent state about the clients. For instance, instead of maintaining a seek offset, the server may require the client to specify the offset for each read or write. Stateful servers are faster, since the server can take advantage of its knowledge of client state to eliminate a lot of network traffic. However, they have complex consistency and crash recovery mechanisms. Stateless servers are simpler to design and implement, but do not yield as good performance.
- **Semantics of sharing** — The distributed file system must define the semantics that apply when multiple clients access a file concurrently. *UNIX semantics* require that changes made by one client be visible to all other clients when they issue the next read or write system call. Some file systems provide *session semantics*, where the changes are propagated to other clients at the open and close system call granularity. Some provide even weaker guarantees, such as a time interval that must elapse before the changes are certain to have propagated to other clients.
- **Remote access methods** — A pure client-server model uses the *remote service method* of file access, wherein each action is initiated by the client, and the server is simply an agent that does the client's bidding. In many distributed systems, particularly stateful ones, the server plays a much more active role. It not only services client requests, but also participates in cache coherency mechanisms, notifying clients whenever their cached data is invalid.

We now look at the distributed file systems that are popular in the UNIX world, and see how they deal with these issues.

10.3 Network File System (NFS)

Sun Microsystems introduced NFS in 1985 as a means of providing transparent access to remote file systems. Besides publishing the protocol, Sun also licensed a reference implementation, which was used by vendors to port NFS to several operating systems. NFS has since become a *de facto* industry standard, supported by virtually every UNIX variant and several non-UNIX systems such as VMS and MS-DOS.

The NFS architecture is based on a *client-server* model. A file server is a machine that exports a set of files. Clients are machines that access such files. A single machine can act as both a server and a client for different file systems. The NFS code, however, is split into client and server portions, allowing client-only or server-only systems.

Clients and servers communicate via *remote procedure calls*, which operate as synchronous requests. When an application on the client tries to access a remote file, the kernel sends a request to the server, and the client process blocks until it receives a reply. The server waits for incoming client requests, processes them, and sends replies back to the clients.

10.3.1 User Perspective

An NFS server *exports* one or more file systems. Each exported file system could be either an entire disk partition or a subtree thereof.[1] The server can specify, typically through entries in the **/etc/exports** file, which clients may access each exported file system and whether the access permitted is read-only or read-write.

Client machines can then mount such a file system, or a subtree of it, onto any directory in their existing file hierarchy, just as they would mount a local file system. The client may mount the directory as read-only, even though the server has exported it as read-write. NFS supports two types of mounts—*hard* and *soft*. This influences client behavior if the server does not respond to a request. If the file system is hard-mounted, the client keeps retrying the request until a reply is received. For a soft-mounted file system, the client gives up after a while and returns an error. Once the *mount* succeeds, the client can access files in the remote file system using the same operations that apply to local files. Some systems also support *spongy mounts*, which behave as hard mounts for mount retries but as soft mounts for subsequent I/O operations.

NFS mounts are less restrictive than those of local file systems. The protocol does not require the caller of mount to be a privileged user, although most clients impose this requirement.[2] The client may mount the same file system at multiple locations in the directory tree, even onto a subdirectory of itself. The server can export only its local file systems and may not cross its own mount points during pathname traversal. Thus, for a client to see the all the files on a server, it must mount all of the server's file systems.

This is illustrated in Figure 10-1. The server system **nfssrv** has two disks. It has mounted **dsk1** on the **/usr/local** directory of **dsk0** and has exported the directories **/usr** and **/usr/local**. Suppose a client executes the following four *mount* operations:

```
mount -t nfs  nfssrv:/usr      /usr
mount -t nfs  nfssrv:/usr/u1   /u1
mount -t nfs  nfssrv:/usr      /users
mount -t nfs  nfssrv:/usr/local  /usr/local
```

All four mounts will succeed. On the client, the **/usr** subtree reflects the entire **/usr** subtree of **nfssrv**, since the client has also mounted **/usr/local**. The **/u1** subtree on the client maps the **/usr/u1** subtree on **nfssrv**. This illustrates that it is legal to mount a subdirectory of an exported file system.[3] Finally, the **/users** subtree on the client only maps that part of the **/usr** subtree of **nfssrv** that resides on **dsk0**; the file system on **dsk1** is not visible under **/users/local** on the client.

10.3.2 Design Goals

The original NFS design had the following objectives:

[1] Different UNIX variants have their own rules governing the granularity of the exports. Some may only allow an entire file system to be exported, whereas some may permit only one subtree per file system.

[2] Digital's ULTRIX, for instance, allows any user to mount an NFS file system so long as the user has write permission to the mount point directory.

[3] Not all implementations allow this.

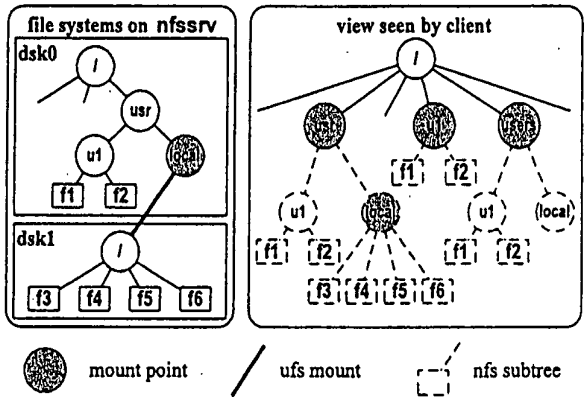


Figure 10-1. Mounting NFS file systems.

- NFS should not be restricted to UNIX. Any operating system should be able to implement an NFS server or client.
- The protocol should not be dependent on any particular hardware.
- There should be simple recovery mechanisms from server or client crashes.
- Applications should be able to access remote files transparently, without using special pathnames or libraries and without recompiling.
- UNIX file system semantics must be maintained for UNIX clients.
- NFS performance must be comparable to that of a local disk.
- The implementation must be transport-independent.

10.3.3 NFS Components

An NFS implementation is composed of several components. Some of these are localized either to the server or to the client, whereas some are shared by both. A few components are not required for the core functionality, but form part of the extended NFS interface:

- The *NFS protocol* defines the set of requests that may be made by the client to the server and the arguments and return values for each. Version 1 of the protocol existed only within Sun Microsystems and was never released. All NFS implementations support *NFS version 2 (NFSv2)*,[4] which was first released in SunOS 2.0 in 1985 [Sand 85b]; hence this chapter deals mainly with this version. Section 10.10 discusses *version 3* of the protocol,

[4] This includes those that support NFSv3.

which was published in 1993 and has been implemented by a number of vendors. Table 10-1 enumerates the complete set of NFSv2 requests.

- The *Remote Procedure Call (RPC) protocol* defines the format of all interactions between the client and the server. Each NFS request is sent as an RPC packet.
- The *Extended Data Representation (XDR)* provides a machine-independent method of encoding data to send over the network. All RPC requests use XDR encoding to pass data. Note that XDR and RPC are used for many other services besides NFS.
- The *NFS server code* is responsible for processing all client requests and providing access to the exported file systems.
- The *NFS client code* implements all client system calls on remote files by sending one or more RPC requests to the server.
- The *Mount protocol* defines the semantics for mounting and unmounting NFS file systems. Table 10-2 contains a brief description of the protocol.
- Several *daemon processes* are used by NFS. On the server, a set of *nfsd* daemons listen for and respond to client NFS requests, and the *mountd* daemon handles mount requests. On the client, a set of *biod* daemons handles asynchronous I/O for blocks of NFS files.
- The *Network Lock Manager (NLM)* and the *Network Status Monitor (NSM)* together provide the facilities for locking files over a network. These facilities, while not formally tied to NFS, are found on most NFS implementations and provide services not possible in the base protocol. NLM and NSM implement the server functionality via the *lockd* and *statd* daemons, respectively.

Table 10-1. NFSv2 operations

Proc	Input args	Results
NULL	void	void
GETATTR	fhandle	status, fattr
SETATTR	fhandle, sattr	status, fattr
LOOKUP	dirfh, name	status, fhandle, fattr
READLINK	fhandle	status, link_value
READ	fhandle, offset, count, totcount	status, fattr, data
WRITE	fhandle, offset, count, totcount, data	status, fattr
CREATE	dirfh, name, sattr	status, fhandle, fattr
REMOVE	dirfh, name	status
RENAME	dirfh1, name1, dirfh2, name2	status
LINK	fhandle, dirfh, name	status
SYMLINK	dirfh, name, linkname, sattr	status
MKDIR	dirfh, name, sattr	status, fhandle, fattr
RMDIR	dirfh, name	status
READDIR	fhandle, cookie, count	status, dir_entries
STATFS	fhandle	status, file_stats

Key: fattr = file attributes; sattr = attributes to set; cookie = opaque object returned by previous READDIR; fhandle = file handle; dirfh = file handle of directory.

Table 10-2. Mount protocol (version 1)

Procedure	Input args	Results
NULL	void	void
MNT	pathname	status, fhandle
DUMP	void	mount list
UMNT	pathname	void
UMNTALL	void	void
EXPORT	void	export list

Key: fhandle = handle of top-level directory of mounted subtree.

10.3.4 Statelessness

The single most important characteristic of the NFS protocol is that the server is stateless and does not need to maintain any information about its clients to operate correctly. Each request is completely independent of others and contains all the information required to process it. The server need not maintain any record of past requests from clients, except optionally for caching or statistics gathering purposes.

For example, the NFS protocol does not provide requests to open or close a file, since that would constitute state information that the server must remember. For the same reason, the READ and WRITE requests pass the starting offset as a parameter, unlike *read* and *write* operations on local files, which obtain the offset from the *open file object* (see Section 8.2.3).[5]

A stateless protocol makes crash recovery simple. No recovery is required when a client crashes, since the server maintains no persistent information about the client. When the client reboots, it may remount the file systems and start up applications that access the remote files. The server neither needs to know nor cares about the client crashing.

When a server crashes, the client finds that its requests are not receiving a response. It then continues to resend the requests until the server reboots.[6] At that time, the server will receive the requests and can process them since the requests did not depend on any prior state information. When the server finally replies to the requests, the client stops retransmitting them. The client has no way to determine if the server crashed and rebooted or was simply slow.

Stateful protocols, however, require complex crash-recovery mechanisms. The server must detect client crashes and discard any state maintained for that client. When a server crashes and reboots, it must notify the clients so that they can rebuild their state on the server.

A major problem with statelessness is that the server must commit all modifications to stable storage before replying to a request. This means that not only file data, but also any metadata such as inodes or indirect blocks must be flushed to disk before returning results. Otherwise, a server crash might lose data that the client believes has been successfully written out to disk. (A system

[5] Some systems provide *pread* and *pwrite* calls, which accept the offset as an argument. This is particularly useful for multithreaded systems (see Section 3.3.2).

[6] This is true only for *hard mounts* (which are usually the default). For *soft mounts*, the client gives up after a while and returns an error to the application.

crash can lose data even on a local file system, but in that case the users are aware of the crash and of the possibility of data loss.) Statelessness also has other drawbacks. It requires a separate protocol (NLM) to provide file locking. Also, to address the performance problems of synchronous writes, most clients cache data and metadata locally. This compromises the consistency guarantees of the protocol, as discussed in detail in Section 10.7.2.

10.4 The Protocol Suite

The primary protocols in the NFS suite are RPC, NFS, and Mount. They all use XDR for data encoding. Other related protocols are the NLM, NSM, and the *portmapper*. This section describes XDR and RPC.

10.4.1 Extended Data Representation (XDR)

Programs that deal with network-based communications between computers have to worry about several issues regarding the interpretation of data transferred over the network. Since the computers at each end might have very different hardware architectures and operating systems, they may have different notions about the internal representation of data elements. These differences include byte ordering, sizes of data types such as integers, and the format of strings and arrays. Such issues are irrelevant for communication with the same machine or even between two like machines, but must be resolved for heterogeneous environments.

Data transmitted between computers can be divided into two categories—opaque and typed. Opaque, or byte-stream, transfers may occur, for example, in file transfers or modem communications. The receiver simply treats the data as a stream of bytes and makes no attempt to interpret it. Typed data, however, is interpreted by the receiver, and this requires that the sender and receiver agree on its format. For instance, a *little-endian* machine may send out a two-byte integer with a value of 0x0103 (259 in decimal). If this is received by a *big-endian* machine, it would (in absence of prior conventions) be interpreted as 0x0301 (decimal 769). Obviously, these two machines will not be able to understand each other.

The XDR standard [Sun 87] defines a machine-independent representation for data transmission over a network. It defines several basic data types and rules for constructing complex data types. Since it was introduced by Sun Microsystems, it has been influenced by the Motorola 680x0 architecture (the Sun-2 and Sun-3 workstations were based on the 680x0 hardware) in issues such as byte ordering. Some of the basic definitions of XDR are as follows:

- **Integers** are 32-bit entities, with byte 0 (numbering the bytes from left to right) representing the most significant byte. Signed integers are represented in two's complement notation.
- **Variable-length opaque data** is described by a *length* field (which is a four-byte integer), followed by the data itself. The data is NULL-padded to a four-byte boundary. The *length* field is omitted for fixed-length opaque data.

- **Strings** are represented by a *length* field followed by the ASCII bytes of the string, NULL-padded to a four-byte boundary. If the string length is an exact multiple of four, there is no (UNIX-style) trailing NULL byte.
- **Arrays** of homogeneous elements are encoded by a *size* field followed by the array elements in their natural order. The size field is a four-byte integer and is omitted for fixed-size arrays. Each element's size must be a multiple of four bytes. While the elements must be of the same type, they may have different sizes, for instance, in an array of strings.
- **Structures** are represented by encoding their components in their natural order. Each component is padded to a four-byte boundary.

Figure 10-2 illustrates some examples of XDR encoding. In addition to this set of definitions, XDR also provides a formal language specification to describe data. The RPC specification language, described in the next section, simply extends the XDR language. Likewise, the *rpcgen* compiler understands the XDR specification and generates routines that encode and decode data in XDR form.

XDR forms a universal language for communication between arbitrary computers. Its major drawback is the performance penalty paid by computers whose natural data representation semantics do not match well with that of XDR. Such computers must perform expensive conversion operations for each data element transmitted. This is most wasteful when the two computers themselves are of like type and do not need data encoding when communicating with each other.

For instance, consider two VAX-11 machines using a protocol that relies on XDR encoding. Since the VAX is little-endian (byte 0 is least significant byte), the sender would have to convert each integer to big-endian form (mandated by XDR), and the receiver would have to convert it back to little-endian form. This wasteful exercise could be prevented if the representation provided some

Value	XDR Representation
0x203040	00 20 30 40
Array of 3 integers {0x30, 0x40, 0x50}	00 00 00 30 00 00 00 40 00 00 00 50

Variable length array of strings
{
"Monday",
"Tuesday"
}
	00 00 00 02
	00 00 00 06
	'M' 'o' 'n' 'd'
	'a' 'y' 00 00
	00 00 00 07
	'T' 'u' 'e' 's'
	'd' 'a' 'y' 00

Figure 10-2. Examples of XDR encoding.

means of communicating the machine characteristics, so that conversions were required only for unlike machines. DCE RPC [OSF 92] uses such an encoding scheme in place of XDR.

10.4.2 Remote Procedure Calls (RPC)

The *Remote Procedure Call (RPC) protocol* specifies the format of communications between the client and the server. The client sends RPC requests to the server, which processes them and returns the results in an RPC reply. The protocol addresses issues such as message format, transmission, and authentication, which do not depend on the specific application or service. Several services have been built on top of RPC, such as NFS, Mount, NLM, NSM, *portmapper*, and Network Information Service (NIS).

> *There are several different RPC implementations. NFS uses the RPC protocol introduced by Sun Microsystems [Sun 88], which is known as Sun RPC or ONC-RPC. (ONC stands for Open Network Computing.) Throughout this book, the term RPC refers to Sun RPC, except when explicitly stated otherwise. The only other RPC facility mentioned in the book is that of OSF's Distributed Computing Environment, which is referred to as DCE RPC.*

Unlike DCE RPC, which provides synchronous and asynchronous operations, Sun RPC uses synchronous requests only. When a client makes an RPC request, the calling process blocks until it receives a response. This makes the behavior of the RPC similar to that of a local procedure call.

The RPC protocol provides reliable transmission of requests, meaning it must ensure that a request gets to its destination and that a reply is received. Although RPC is fundamentally transport-independent, it is often implemented on top of UDP/IP (User Datagram Protocol/Internet Protocol), which is inherently unreliable. The RPC layer implements a reliable datagram service by keeping track of unanswered requests and retransmitting them periodically until a response is received.

Figure 10-3 describes a typical RPC request and (successful) reply. The `xid` is a transmission ID, which tags a request. The client generates a unique `xid` for each request, and the server returns the same `xid` in the reply. This allows the client to identify the request for which the response has arrived and the server to detect duplicate requests (caused by retransmissions from the client). The `direction` field identifies the message as a request or a reply. The `rpc_vers` field identifies the version number of the RPC protocol (current version = 2). `prog` and `vers` are the program and version number of the specific RPC service. An RPC service may register multiple protocol versions. The NFS protocol, for instance, has a program number of 100003 and supports version numbers 2 and 3. `proc` identifies the specific procedure to call within that service program. In the reply, the `reply_stat` and `accept_stat` fields contain status information.

RPC uses five authentication mechanisms to identify the caller to the server—`AUTH_NULL`, `AUTH_UNIX`, `AUTH_SHORT`, `AUTH_DES`, and `AUTH_KERB`. `AUTH_NULL` means no authentication. `AUTH_UNIX` is composed of UNIX-style credentials, including the client machine name, a UID, and one or more GIDs. The server may generate an `AUTH_SHORT` upon receipt of an `AUTH_UNIX` credential, and return it to the caller for use in subsequent requests. The idea is that the server can decipher `AUTH_SHORT` credentials very quickly to identify known clients, thus providing faster authentication. This is an optional feature and not many services support it. `AUTH_DES` is a secure authentication facility using a mechanism called *private keys* [Sun 89]. `AUTH_KERB` is another secure facility based on the *Kerberos* authentication mechanism [Stei 88]. Each service decides which authentication mechanisms to accept. NFS allows all five, except that it allows `AUTH_NULL` only for the `NULL` procedure. Most NFS implementations, however, use `AUTH_UNIX` exclusively.

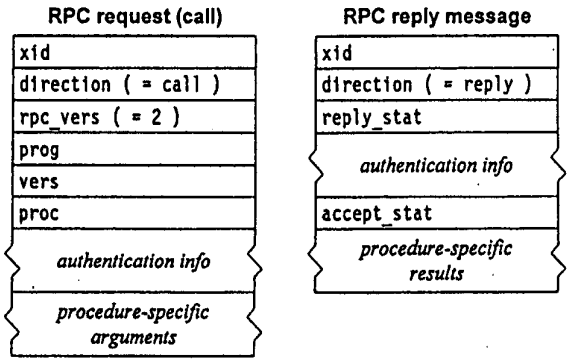


Figure 10-3. RPC message formats.

Sun also provides an RPC programming language, along with an *RPC compiler* called *rpcgen*. An RPC-based service can be fully specified in this language, resulting in a formal interface definition. When *rpcgen* processes this specification, it generates a set of *C* source files containing XDR conversion routines and stub versions of the client and server routines, and a header file containing definitions used by both client and server.

10.5 NFS Implementation

We now examine how typical UNIX systems implement the NFS protocol. NFS has been ported to several non-UNIX systems such as MS-DOS and VMS. Some of these are client-only or server-only implementations, while others provide both pieces. Moreover, there are several dedicated NFS server implementations from vendors like Auspex, Network Appliance Corporation, and Novell, which do not run on general-purpose operating systems. Finally, there are a number of user-level implementations of NFS for various operating systems, many available as shareware or freeware. The implementation details on such systems are, of course, fundamentally different, and Section 10.8 describes some interesting variations. In this section, we restrict our discussion to kernel implementations of NFS in conventional UNIX systems that also support the vnode/vfs interface.

10.5.1 Control Flow

In Figure 10-4, the server has exported a *ufs* file system, mounted by the client. When a process on the client makes a system call that operates on an NFS file, the file-system-independent code identifies the vnode of the file and invokes the relevant vnode operation. For NFS files, the file-system-dependent data structure associated with the vnode is the *rnode* (for remote node). The `v_op` field in the vnode points to the vector of NFS client routines (`struct vnodeops`) that implement the various vnode operations. These routines act by constructing RPC requests and sending them to the server. The calling process on the client blocks until the server responds (see footnote 6). The server processes the requests by identifying the vnode for the corresponding local file and invoking the appropriate `vnodeops` calls, which are implemented by the local file system (*ufs* in this example).

Finally, the server completes processing the request, bundles the results into an RPC reply message and sends it back to the client. The RPC layer on the client receives the reply and wakes up the sleeping process. This process completes execution of the client vnode operation and the rest of the system call, and returns control to the user.

This implies the client must maintain a local vnode for each active NFS file. This provides the proper linkages to route the vnode operations to the NFS client functions. It also allows the client to cache attributes of remote files, so that it can perform some operations without accessing the server. The issues related to client caching are discussed in Section 10.7.2.

10.5.2 File Handles

When a client makes an NFS request to the server, it must identify the file it wants to access. Passing the pathname on each access would be unacceptably slow. The NFS protocol associates an object called a *file handle* with each file or directory. The server generates this handle when the client first accesses or creates the file through a `LOOKUP`, `CREATE`, or `MKDIR` request. The server returns the handle to the client in the reply to the request, and the client can subsequently use it in other operations on this file.

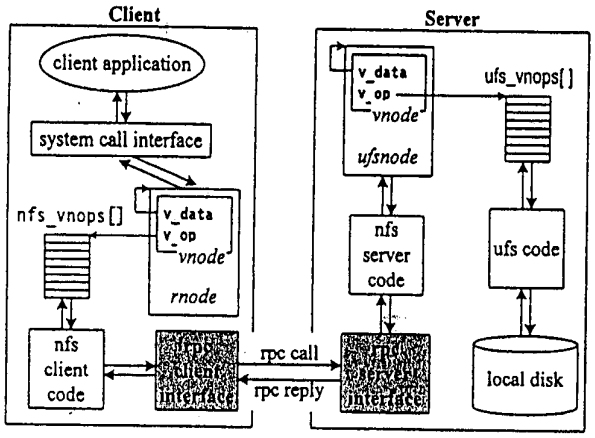


Figure 10-4. Control flow in NFS.

The client sees the file handle as an opaque, 32-byte object and makes no attempt to interpret the contents. The server can implement the file handle as it pleases, as long as it provides a unique one-to-one mapping between files and handles. Typically, the file handle contains a *file system ID*, which uniquely identifies the local file system, the *inode number* of the file, and the *generation number* of that inode. It may also contain the inode number and generation number of the exported directory through which the file was accessed.

The generation number was added to the inode to solve problems peculiar to NFS. It is possible that, between the client's initial access of the file (typically through `LOOKUP`, which returns the file handle) and when the client makes an I/O request on the file, the server deletes the file and reuses its inode. Hence the server needs a way of determining that the file handle sent by the client is obsolete. It does this by incrementing the generation number of the inode each time the inode is freed (the associated file is deleted). The server can now recognize requests that refer to the old file and respond to them with an `ESTALE` *(stale file handle)* error status.

10.5.3 The Mount Operation

When a client mounts an NFS file system, the kernel allocates a new `vfs` structure and invokes the `nfs_mount()` function. `nfs_mount()` sends an RPC request to the server, using the Mount protocol. The argument to this request is the pathname of the directory to be mounted. The *mountd* daemon on the server receives this request and translates the pathname. It checks whether the pathname is that of a directory and whether that directory is exported to the client. If so, *mountd* returns a successful completion reply, in which it sends the file handle of that directory.

The client receives the successful reply and completes the initialization of the `vfs` structure. It records the name and network address of the server in the private data object of the *vfs*. It then allocates the rnode and vnode for the root directory. If the server crashes and reboots, the clients still have the file systems mounted, but the server has lost that information. Since the clients are sending valid file handles in NFS requests, the server can assume that a successful *mount* had taken place previously and rebuild its internal records.

The server does, however, need to check access rights to the file system on *each* NFS request. It must ensure that the files being operated on are exported to that client (if the request will modify the file, it must be exported read-write). To make this check efficient, the file handle contains the *<inode, generation number>* of the exported directory. The server maintains an in-memory list of all exported directories, so it can perform this check quickly.

10.5.4 Pathname Lookup

The client gets the file handle of the top-level directory as a result of the *mount* operation. It obtains other handles during pathname lookup or as a result of `CREATE` or `MKDIR`. The client initiates *lookup* operations during system calls such as *open*, *creat*, and *stat*.

On the client, *lookup* begins at the current or root directory (depending on whether the pathname is relative or absolute) and proceeds one component at a time. If the current directory is an NFS directory or if we cross a mount point and get to an NFS directory, the lookup operation calls the NFS-specific VOP_LOOKUP function. This function sends a LOOKUP request to the server, passing it the file handle of the parent directory (which the client had saved in the rnode) and the name of the component to be searched.

The server extracts the file system ID from the handle and uses it to locate the vfs structure for the file system. It then invokes the VFS_VGET operation on this file system, which translates the file handle and returns a pointer to the vnode of the parent directory (allocating a new one if it is not already in memory). The server then invokes the VOP_LOOKUP operation on that vnode, which calls the corresponding function of the local file system. This function searches the directory for the file and, if found, brings its vnode into memory (unless it is already there, of course) and returns a pointer to it.

The server next invokes the VOP_GETATTR operation on the vnode of the target file, followed by VOP_FID, which generates the file handle for the file. Finally, it generates the reply message, which contains the status, the file handle of the component, and its file attributes.

When the client receives the reply, it allocates a new rnode and vnode for the file (if this file had been looked up previously, the client may already have a vnode for it). It copies the file handle and attributes into the rnode and proceeds to search for the next component.

Searching for one component at a time is slow and requires several RPCs for a single pathname. The client may avoid some RPCs by caching directory information (see Section 10.7.2.). Although it would seem more efficient to send the entire pathname to the server in a single LOOKUP call, that approach has some important drawbacks. First, since the client may have mounted a file system on an intermediate directory in the path, the server needs to know about all the client's mount points to parse the name correctly. Second, parsing an entire pathname requires the server to understand UNIX pathname semantics. This conflicts with the design goals of statelessness and operating system independence. NFS servers have been ported to diverse systems such as VMS and Novell NetWare, which have very different pathname conventions. Complete pathnames are used only in the *mount* operation, which uses a different protocol altogether.

10.6 UNIX Semantics

Since NFS was primarily intended for UNIX clients, it was important that UNIX semantics be preserved for remote file access. The NFS protocol, however, is stateless, which means that clients cannot maintain open files on the server. This leads to a few incompatibilities with UNIX, which we describe in the following paragraphs.

10.6.1 Open File Permissions

UNIX systems check access permissions when a process first opens the file, not on every read or write. Suppose, after a user opens a file for writing, the owner of the file changes its permissions to read-only. On a UNIX system, the user can continue to write to the file until he closes it. NFS,

lacking the concept of open files, checks the permissions on each read or write. It would therefore return an error in such a case, which the client would not expect to happen.

Although there is no way to fully reconcile this issue, NFS provides a work-around. *The server always allows the owner of the file to read or write the file, regardless of the permissions.* On the face of it, this seems a further violation of UNIX semantics, since it appears to allow owners to modify their own write-protected files. The NFS client code, however, prevents that from happening. When the client opens the file, the LOOKUP operation returns the file attributes along with the handle. The attributes contain the file permissions at the time of the *open*. If the file is write-protected, the client code returns an EACCESS *(access denied)* error from the *open* call. It is important to note that the server's security mechanisms rely on proper behavior of the client. In this instance, the problem is not serious, since it only affects the owner's lack of write access. Section 10.9 discusses the major problems with NFS security.

10.6.2 Deletion of Open Files

If a UNIX process deletes a file that another process (or that process itself) still has open, the kernel simply marks the file for deletion and removes its entry from the parent directory. Although no new processes can now open this file, those that have it open can continue to access it. When the last process that has the file open closes it, the kernel physically deletes the file. This feature is used by several utilities to implement temporary files.

Once again, this is a problem for NFS, since the server does not know about open files. The work-around this time involves modifying the NFS client code, since the client *does* know about open files. When the client detects an attempt to delete an open file, it changes the operation to a RENAME, giving the file a new location and name. The client usually chooses some unusual and long name that is unlikely conflict with existing files. When the file is last closed, the client issues a REMOVE request to delete the file.

This works well when the process that has the file open is on the same machine as the one deleting the file. There is no protection, however, against the file being deleted by a process on another client (or on the server). If that happens, the user will get an unexpected error *(stale file handle)* when he next tries to read or write that file. Another problem with this work-around is that if the client crashes between the RENAME and REMOVE operations, a garbage file is left behind on the server.

10.6.3 Reads and Writes

In UNIX, a *read* or *write* system call locks the vnode of the file at the start of the I/O. This makes file I/O operations atomic at the system call granularity. If two *writes* are issued to the same file at roughly the same time, the kernel serializes them and completes one before starting the other. Likewise, it ensures that the file cannot change while a *read* is in progress. The local file-system-dependent code handles the locking, within the context of a single vop_rdwr operation.

For an NFS file, the client code serializes concurrent access to a file by two processes on the same client. If, however, the two processes are on different machines, they access the server independently. A *read* or *write* operation may span several RPC requests (the maximum size of an RPC

message is 8192 bytes), and the server, being stateless, maintains no locks between requests. NFS offers no protection against such overlapping I/O requests.

Cooperating processes can use the Network Lock Manager (NLM) protocol to lock either entire files or portions thereof. This protocol only offers advisory locking, which means that another process can always bypass the locks and access the file if it so chooses.

10.7 NFS Performance

One of the design goals of NFS was that its performance be comparable to that of a small local disk. The metric of interest is not raw throughput, but the time required to do normal work. There are several benchmarks that try to simulate a normal workload on an NFS file system, the most popular being *LADDIS* [Witt 93] and *nhfsstone* [Mora 90]. This section discusses the major performance problems of NFS and how they have been addressed.

10.7.1 Performance Bottlenecks

NFS servers are usually powerful machines with large caches and fast disks, which compensate for the time taken for RPC requests going back and forth. There are several areas, however, where the NFS design directly leads to poor performance.

Being a stateless protocol, NFS requires that all writes be committed to stable storage before replying to them. This includes not only modifications to the file metadata (inodes and indirect blocks), but also to the body of the file. As a result, any NFS request that modifies the file system in any way (such as `WRITE`, `SETATTR`, or `CREATE`) is extremely slow.

Fetching of file attributes requires one RPC call per file. As a result, a command such as *ls -l* on a directory results in a large number of RPC requests. In the local case, such an operation is fast, since the inodes end up in the buffer cache, so the *stat* calls need only a memory reference.

If the server does not reply to a request fast enough, the client retransmits it to account for the server crashing or the request being lost on the network. Processing the retransmitted request further adds to the load on the server and may aggravate the problem. This has a cascading effect, resulting in an overloaded server bogged down by the incoming traffic.

Let us look at some ways of addressing NFS performance problems and the repercussions of these solutions.

10.7.2 Client-Side Caching

If every operation on a remote file required network access, NFS performance would be intolerably slow. Hence most NFS clients resort to caching both file blocks and file attributes. They cache file blocks in the buffer cache and attributes in the modes. This caching is dangerous, since the client has no way of knowing if the contents of the cache are still valid, short of querying the server each time they must be used.

Clients take certain precautions to reduce the dangers of using stale data. The kernel maintains an expiry time in the mode, which monitors how long the attributes have been cached. Typically, the client caches the attributes for 60 seconds or less after fetching them from the server. If

they are accessed after the quantum expires, the client fetches them from the server again. Likewise, for file data blocks, the client checks cache consistency by verifying that the file's *modify time* has not changed since the cached data was read from the server. The client may use the cached value of this timestamp or issue a `GETATTR` if it has expired.

Client-side caching is essential for acceptable performance. The precautions described here reduce, but do not eliminate, the consistency problems. In fact, they introduce some new race conditions, as described in [Mack 91] and [Jusz 89].

10.7.3 Deferral of Writes

The NFS requirement of synchronous writes applies only to the server. The client is free to defer writes, since if data is lost due to a client crash, the users know about it. The client policy, therefore, is to use asynchronous writes for full blocks (issue the `WRITE` request but do not wait for the reply) and delayed writes for partial blocks (issue the `WRITE` sometime later). Most UNIX implementations flush delayed writes to the server when the file is closed and also every 30 seconds. The *biod* daemons on the client handle these writes.

Although the server must commit writes to stable storage before replying, it does not have to write them to disk. It may use some special hardware to make sure that the data will not be lost in the event of a crash. For instance, some servers use a special, battery backed, *nonvolatile memory (NVRAM)*. The `WRITE` operation simply transfers the data to an NVRAM buffer (provided one was free). The server flushes the NVRAM buffers to disk at a later time. This allows the server to respond quickly to write requests, since the transfer to NVRAM is much faster than a disk write. The disk driver can optimize the order of the NVRAM-to-disk writes, so as to minimize disk head movements. Moreover, multiple updates to the same buffer could be written to disk in a single operation. [Mora 90] and [Hitz 94] describe NFS server implementations that use NVRAM.

[Jusz 94] shows a technique called write-gathering that reduces the synchronous write bottleneck without using special hardware. It relies on the fact that typical NFS clients use a number of *biod* daemons to handle write operations. When a client process opens a file and writes to it, the kernel simply caches the changes and marks them for delayed write. When the client closes the file, the kernel flushes its blocks to the server. If there are sufficient *biod*s available on the client, they can all issue the writes in parallel. As a result, servers often receive a number of writes requests for the same file bunched together.

Using write-gathering, the server does not process `WRITE` requests immediately. Rather, it delays them for a little while, in the hope that it receives other `WRITE`s for the same file in the meantime. It then gathers all requests for the same file and processes them together. After completing them all, the server replies to each of them. This technique is effective when the clients use *biod*s and is optimal when they use a large number of *biod*s. Although it appears to increase the latency of individual requests, it improves performance tremendously by reducing the total number of disk operations on the server. For instance, if the server gathers *n* writes to a file, it may be able to commit them to disk using a single data write and a single inode write (as opposed to *n* of each). Write-gathering improves the server throughput, and reduces the load on its disks. These gains more than offset the latency increase caused by the delay, and the overall effect is to reduce the average time of the writes.

Some servers rely on an *uninterruptible power supply (UPS)* to flush blocks to disk in case of a crash. Some simply ignore the NFS requirement of synchronous writes, expecting crashes to be rare occurrences. The plethora of solutions and work-arounds to this problem simply highlights its severity. NFSv3, described in Section 10.10, provides a protocol change that allows clients and servers to use asynchronous writes safely.

10.7.4 The Retransmissions Cache

In order to provide reliable transmission, RPC clients repeatedly retransmit requests until they receive a response. Typically, the waiting period is quite short (configurable, typically 1–3 seconds) for the first retransmission and increases exponentially for each subsequent retry. If, after a certain number of retries, the client still does not receive a response, it may (in some implementations) send a new request identical to the old, but with a different transmission ID *(xid)*.

Retransmissions occur due to packet loss (original request or the reply) on the network or because the server could not reply promptly enough. Very often the reply to the original request is on the way when the client sends a second copy. Multiple retransmissions usually happen when the server crashes or when the network is extremely congested.

The server needs to handle such duplicate requests correctly and efficiently. NFS requests can be divided into two types—*idempotent* and *nonidempotent* [Jusz 89]. Idempotent requests, such as `READ` or `GETATTR`, can be executed twice without any ill effects. Nonidempotent requests may result in incorrect behavior if repeated. Requests that modify the file system in any way are nonidempotent.

For example, consider the following sequence of events caused by a duplicate `REMOVE` operation:

1. Client sends a `REMOVE` request for a file.
2. Server removes the file successfully.
3. Server sends a success reply to the client. This reply is lost on the network.
4. Client sends a duplicate `REMOVE` request.
5. Server processes the second `REMOVE`, which fails because the file has already been deleted.
6. Server sends an error message to the client. This message reaches the client.

As a result, the client gets an error message, even though the `REMOVE` operation succeeded.

Reprocessing of duplicate requests also hurts server performance, because the server spends a lot of time doing work it should never do. This aggravates a situation that was bad to start with, since the retransmission probably occurred because the server was overloaded and therefore slow.

It is therefore necessary to detect and handle retransmissions correctly. To do so, the server keeps a cache of recent requests. A request can be identified as a duplicate if its xid, procedure number, and client ID match those of another request in the cache. (This is not always foolproof, because some clients may generate the same xid for requests from two different users.) This cache is normally called the *retransmissions cache* or the *xid cache*.

The original Sun reference port maintained such a cache only for the `CREATE`, `REMOVE`, `LINK`, `MKDIR`, and `RMDIR` requests. It checked the cache only after a request failed, to determine if the failure was due to the request being a retransmission. If so, it sent a success reply to the client. This was

inadequate—it covered only some of the loopholes and opened up new consistency problems. Moreover, it does not address the performance problems, since the server does not check the cache until after it has processed the request.

[Jusz 89] provides a detailed analysis of the problems in handling retransmissions. Based on that, Digital revamped the xid cache in Ultrix. The new implementation caches *all* requests and checks the cache before processing new requests. Each cache entry contains request identification (client ID, xid, and procedure number), a *state* field, and a *timestamp*. If the server finds the request in the cache, and the state of the request is *in progress*, it simply discards the duplicate. If the state is *done*, the server discards the duplicate if the timestamp indicates that the request has completed recently (within a *throwaway window* set at 3–6 seconds). Beyond the throwaway window, the server processes the request if idempotent. For nonidempotent requests, the server checks to see if the file has been modified since the original timestamp. If not, it sends a success response to the client; otherwise, it retries the request. Cache entries are recycled on a *least recently used* basis so that if the client continues to retransmit, the server will eventually process the request again.

The xid cache helps eliminate several duplicate operations, improving both the performance and the correctness of the server. It is possible to take this one step further. If the server caches the reply message along with the xid information, then it can handle a duplicate request by retransmitting the cached reply. Duplicates that arrive within the throwaway window may still be discarded altogether. This will further reduce wasteful reprocessing, even for idempotent requests. This approach requires a large cache capable of saving entire reply messages. Some replies, such as those for `READ` or `READDIR` requests, can be very large. It is better to exclude these requests from the xid cache and process them again if necessary.

(下頁)

10.8 Dedicated NFS Servers

Most UNIX vendors design file servers by repackaging workstations in a rack and adding more disks, network adapters, and memory. These systems run the vendor's UNIX variant, which is designed primarily for use in a multiprogramming environment and is not necessarily appropriate for high-throughput NFS service. Some vendors have designed systems specifically for use as dedicated NFS servers. These systems either add to the functionality of UNIX or completely replace the operating system. This section describes two such architectures.

10.8.1 The Auspex Functional Multiprocessor Architecture

Auspex Systems entered the high-end NFS server market with a functional multiprocessing (FMP) system called the NS5000. The FMP architecture [Hitz 90] recognizes that NFS service comprises two major subsystems—network and file system.[7] It uses a number of Motorola 680x0 processors sharing a common backplane, each dedicated to one of these subsystems. These processors run a small *functional multiprocessing kernel (FMK)* and communicate with each other via high-speed

[7] The original design separated the storage functionality into a third subsystem. The recent line of products do not feature separate storage processors.

message passing. One processor (also a 68020) runs a modified version of SunOS4.1 (with FMK support added to it) and provides management functionality. Figure 10-5 shows the basic design.

The UNIX front end can communicate directly to each of the functional processors. It talks to network processors through the standard network *if* driver and to file system processors through a special *local file system,* which implements the *vfs* interface. The UNIX processor also has direct access to the storage through a special device driver that represents an *Auspex disk* and converts disk I/O requests into FMK messages. This allows utilities such as *fsck* and *newfs* to work without change.

Normal NFS requests bypass the UNIX processor altogether. The request comes in at a network processor, which implements the IP, UDP, RPC, and NFS layers. It then passes the request to the file system processor, which may issue I/O requests to the storage processor. Eventually, the network processor sends back the reply message to the client.

The FMK kernel supports a small set of primitives including light-weight processes, message passing, and memory allocation. By eliminating a lot of the baggage associated with the traditional UNIX kernel, FMK provides extremely fast context switching and message passing. For instance, FMK has no memory management and its processes never terminate.

This architecture provided the basis for a high-throughput NFS server that established Auspex Systems as a leader in the high-end NFS market. Recently, its position has been challenged by cluster-based NFS servers from vendors such as Sun Microsystems and Digital Equipment Corporation.

10.8.2 IBM's HA-NFS Server

[Bhid 91] describes a prototype implementation of HA-NFS, a highly available NFS server designed at IBM. HA-NFS separates the problem of high availability NFS service into three compo-

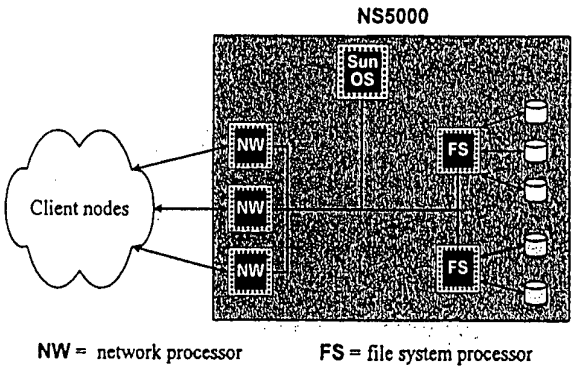


Figure 10-5. Auspex NS5000 architecture.

nents—network reliability, disk reliability, and server reliability. It uses disk mirroring and optional network replication to address the first two problems and uses a pair of cooperating servers to provide server reliability.

Figure 10-6 illustrates the HA-NFS design. Each server has two network interfaces and, correspondingly, two IP addresses. A server designates one of its network interfaces as the primary interface and uses it for normal operation. It uses the secondary interface only when the other server fails.

HA-NFS also uses dual-ported disks, which are connected to both servers through a shared SCSI bus. Each disk has a primary server, which alone accesses it during normal operation. The secondary server takes over the disk when the primary server fails. Thus the disks are divided into two groups, one for each server.

The two servers communicate with each other through periodic heartbeat messages. When a server does not receive a heartbeat from the other, it initiates a series of probes to make sure the other server has actually failed. If so, it initiates a failover procedure. It takes control of the failed server's disks and sets the IP address of its secondary network interface to that of the failed server's primary interface. This allows it to receive and service messages intended for the other server.

The takeover is transparent to clients, who only see reduced performance. The server seems to be unresponsive while the failover is in progress. Once failover completes, the surviving server may be slow, since it now handles the load normally meant for two servers. There is, however, no loss in service.

Each server runs IBM's AIX operating system, which uses a metadata logging file system. HA-NFS adds information about the RPC request to log entries for NFS operations. When a server takes over a disk during failover, it replays the log to restore the file system to a consistent state and recovers its retransmission cache from the RPC information in the log. This prevents inconsistencies due to retransmissions during failover. The two servers also exchange information about clients that

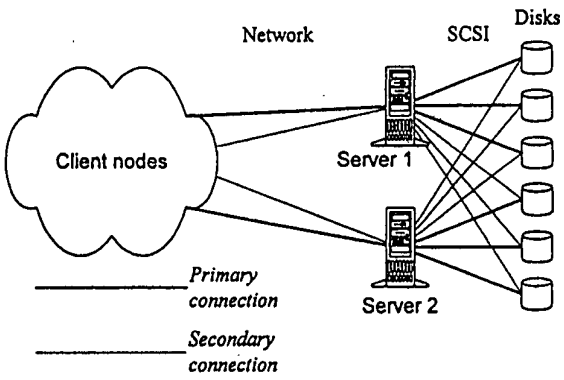


Figure 10-6. HA-NFS configuration.

have made file locking requests using NSM and NLM [Bhid 92]. This allows recovery of the lock manager state after failure of one of the servers.

There are two ways to make the IP address takeover transparent to clients. One is to use special network interface cards that allow their hardware addresses to be changed. During failover, the server changes both the IP address and the hardware address of the secondary interface to those of the failed server's primary interface. In absence of such hardware, the server can take advantage of some side effects of certain *address resolution protocol (ARP)* requests [Plum 82] to update the new *<hardware address, IP address>* mapping in the clients.

10.9 NFS Security

It is difficult for any network application to provide the same level of security as a local system. While NFS seeks to provide UNIX semantics, its security mechanisms are woefully inadequate. This section examines the major loopholes in NFS security and some possible solutions.

10.9.1 NFS Access Control

NFS performs access control at two points—when the file system is mounted and also on every NFS request. Servers maintain an *exports list* that specifies which client machines can access each exported file system and whether the access permitted is read-only or read-write. When a client tries to mount a file system, the server *mountd* checks this list and denies access to ineligible clients. There are no restrictions on specific users, meaning that any user on an eligible client can mount the file system.[8]

On each NFS request, the client sends authentication information, usually in `AUTH_UNIX` form. This information contains the user and group IDs of the owner of the process making the request. The NFS server uses this information to initialize a credentials structure, which is used by the local file systems for access control.

For this to work, the server and all the clients must share a flat *<UID, GID>* space. This means that any given UID must belong to the same person on all machines sharing NFS file systems. If user **u1** on machine **m1** has the same UID as user **u2** on machine **m2**, the NFS server will not be able to distinguish the two and will allow each user complete access to the other's files.

This is a major problem in today's typical workgroups, where each user has his or her own private workstation, and a central NFS server maintains common files (including, in many cases, login directories). Since each user typically has root privileges to his own workstation, he can create accounts with any *<UID, GIDs>*, thereby impersonating anyone else. Such an impostor can freely access the victim's files on the server, without the victim ever learning about it. The impostor can do this without writing sophisticated programs or modifying the kernel or the network. The only line of defense is to restrict NFS access to known clients that can be trusted or monitored. This is the strongest demonstration of the fact that NFS security is nonexistent.

There are other ways of breaking into NFS. Since NFS relies on data that is being sent over unsecured networks, it is easy to write programs that imitate NFS clients and send packets containing fake authentication data, perhaps even appearing to come from a different machine. This would allow break-ins even by users who do not have root permission on a machine or by users on machines that do not have NFS access to a server.

[8] Many client implementations allow only privileged users to mount NFS file systems.

10.9.2 UID Remapping

There are some ways to prevent such intrusions. The first line of defense involves *UID remapping*. This means that instead of a flat *<UID, GIDs>* space on the server and all clients, the server maintains a translation map for each client. This map defines the translation from credentials received over the network to an identity to be used on the server. This identity is also described by a *<UID, GIDs>* set, but the set may be different from that sent by the client.

For instance, the translation map may specify that UID 17 from client **c1** translates to UID 33 on the server, whereas UID 17 from client **c2** translates to UID 17 on the server, and so forth. It may also define translations for the GIDs. The map may contain several default translations or *wildcards* (e.g., *"no translation required for credentials from a set of trusted clients"*). Typically, it would also specify that if a particular incoming credential does not match any map entry or default rule, then that credential is mapped to user *nobody*, which is a special user that can only access files that have world permissions.

Such UID remapping can be implemented at the RPC level. This means that the translations would apply to any RPC-based service and take place before the request was interpreted by NFS. Since such translations require additional processing, this would degrade the performance of all RPC-based services, even if a particular service did not need that level of security.

An alternative approach is to implement UID remapping at the NFS level, perhaps by merging the map with the **/etc/exports** file. This would enable the server to apply different mappings for different file systems. For instance, if the server exports its **/bin** directory as read-only and the **/usr** directory as read-write, it may want to apply UID maps only to the **/usr** directory, which might contain sensitive files that users wish to protect. The drawback of this approach is that each RPC service would have to implement its own UID maps (or other security mechanisms), perhaps duplicating effort and code.

Very few mainstream NFS implementations provide any form of UID remapping at all. Secure versions of NFS prefer to use secure RPC with `AUTH_DES` or `AUTH_KERB` authentication, described in Section 10.4.2.

10.9.3 Root Remapping

A related problem involves root access from client machines. It is not a good idea for superusers on all clients to have root access to files on the server. The usual approach is for servers to map the superuser from any client to the user *nobody*. Alternatively, many implementations allow the **/etc/exports** file to specify an alternative UID to which root should be mapped.

Although this takes care of the obvious problems with superuser access, it has some strange effects on users logged in as root (or who are executing a privileged program installed in *setuid* mode). These users have fewer privileges to NFS files than ordinary users. They may not, for instance, be able to access their own files on the server.